502 052 02

Process-Pascal

Version 4

Users Manual

GB *May 1999*

PROCES-DATA A/S, Navervej 8, DK-8600 Silkeborg, Denmark, Phone +45 87 200 300, Fax + 45 87 200 301

© Copyright 1999 by PROCES-DATA A/S. All rights reserved.

PROCES-DATA A/S reserves the right to make any changes without prior notice.

P-NET, **Soft-Wiring** and **Process-Pascal** are registered trademarks of PROCES-DATA A/S.

CONTENTS **Page**

1 Introduction to Process-Pascal. 1

2 Program Structure in Process-Pascal 2

2.1 Task, an Introduction 3
2.2 Task types. 4
2.3 How to Split a Program into Manageable Tasks. 5

3 Defining Data. 6

3.1 Variables 6
3.2 Identifiers 6

4 Simple Data Types 7

4.1 Ordinal types 7
4.2 The type BOOLEAN. 8
4.3 The type CHAR 8
4.4 The type INTEGER. 9
4.5 The type REAL 10
4.6 The type TIMER 10

5 Structured Types 11

6 Variable Declaration 12

6.1 Global variables 12
6.2 Variables at P-NET 13
6.3 Config 16
6.4 Indirect variables 16

7 Pointer Types 19

8 Constants 20

9 Comments 21

10 Expressions and Assignments 22

10.1 Expressions 22
10.2 Operators 22
10.3 Arithmetic operators 22
10.4 Logical operators 23
10.5 Relational operators 23
10.6 String operator 24
10.7 Operator precedence 25

11 Statements 26

11.1 Simple statements 26
11.2 Assignment 26
11.3 Procedure statement 26
11.4 Structured statements 27
11.5 Compound statement (begin end) 27
11.6 Conditional statement (if then else) 27
11.7 Conditional statement (case) 28
11.8 While statement 28
11.9 Repeat statement 29

11.10 For statement ... 30
11.11 Loop statement ... 30

12 Array ... 31

12.1 One dimensional arrays ... 31
12.2 Multidimensional arrays ... 32

13 Record ... 33

13.1 Variant part ... 33
13.2 Accessing fields ... 34

14 Interface ... 35

14.1 Accessing fields ... 36

15 Buffer ... 37

16 String ... 38

17 Bitmap ... 39

17.1 The smallbitmap type ... 39
17.2 The largebitmap type ... 40
17.3 The videobitmap type ... 40

18 Set ... 41

19 Userdefined Types ... 42

19.1 Subrange types ... 42
19.2 Enumerated types ... 42

20 Structured Constants ... 44

20.1 Array constants ... 44
20.2 Record constants ... 45

21 Procedures and Functions ... 46

21.1 Procedures ... 46
21.2 Value parameters ... 47
21.3 Variable parameters ... 48
21.4 Functions ... 50

22 Scope ... 51

23 Task Declaration ... 52

24 Interrupt ... 56

25 WHEN ERROR ... 58

25.1 WHEN ERROR THEN [Disable] ... 59
25.2 ERROR REPORT ... 61
25.3 ERRORCODES ... 63

26 The SoftWire List ... 64

26.1 The aim of the SoftWire list. ... 64

27 Screen Setup and Definition. ... 65

28 Writing on the Screen. ... 67

29 Keyboard. 72

30 Real-time Clock and Calendar. 73

31 Accessing not Declared Variables. 74

32 PD GATEWAY. 77

33 Modules In Process-Pascal. 81

34 Process-Pascal Reference Lookup. 84

34.1 Standard procedures. 84
34.2 Standard functions 104
34.3 Standard constants 109

35 Comparing Process-Pascal ver. 4.0 to ISO 7185 Standard Pascal. 110

35.1 Exceptions to ISO 7185 STANDARD PASCAL. 110
35.2 Extensions to ISO 7185 Standard Pascal. 111
35.3 Standard Procedures and Functions 112
35.4 Reserved words in Process-Pascal 113
35.5 Compiler directives 114

36 Restrictions in Using Process-Pascal. 116

37 Error Messages and Codes 117

38 Syntax Diagrams 128

39 Index 139

1 Introduction to Process-Pascal.

Process-Pascal is a high level programming language based on Standard Pascal.

Process-Pascal is extended from Standard Pascal with a number of facilities, which make it possible to execute several programmes simultaneously in one computer. This is called multi-tasking.

Process-Pascal is specially developed for use in connection with P-NET, which is a local area network for transmission of data in distributed data acquisition and process control plants. Data, which are distributed in modules on the P-NET, can be defined as variables in Process-Pascal.

Process-Pascal permits automatic reports of alarms in case of error in the controller or in the interface modules. Furthermore, it is possible to automatically test all components of the plant during the starting phase.

Process-Pascal includes standard routines for interactive screen dialogue. Thus it is possible to define that a variable must be shown on the screen and be continuously updated. Data can be keyed into a variable by pointing at it using the screen cursor.

Process-Pascal programmes can be written using any general-purpose editor working with ASCII or ANSI files.

Process-Pascal programmes operate with several types of variables. The compiler automatically performs typecasting during compilation. This makes programmes more safe and easier to develop.

The Process-Pascal program suite provides a debugger, which is a very powerful tool that significantly speeds up the entire process of an application development.

Process-Pascal programmes are compiled with a cross-compiler running on a PC under Windows 95 or Windows NT. The compiler generates code, P-code that is stored in the controller in FLASH or RAM memory. The operating system in the controller interprets the P-codes and executes a piece of machine code for each P-code.

Process-Pascal programmes can not be executed on a PC. The compiler is entirely developed by PROCES-DATA A/S.

This manual is written for programmers who are familiar with Pascal programmes and know about the basic structures for data and programmes.

2 Program Structure in Process-Pascal

Every Process-Pascal program consists of a heading and a block. The structure is illustrated below:

```
Program name [objecttype, capabilities];

VAR
    global variable

 Procedure
    global procedure

Function
    global function

Task name1;
VAR
    local variable
Begin

End;

Task name2;
VAR
    local variable
Begin

End;

Task name3;
VAR
    local variable
Begin

End;

End.
```

Program Heading.

Program heading consists of the word **Program** and a name. Furthermore it contains an objecttype and a list of capabilities for the controller.

Global variable declaration.

External variables in other modules accessed via the P-NET are declared with an identifier, which is used within the program.

Internal variables used to exchange data between tasks, as well internal as external in other controllers.

Global procedure and function declaration.

Global procedures and functions can be called from all internal tasks.
A global procedure/function can be called simultaneously from several tasks with different sets of parameters.

Task declaration.

Program declaration for task. Tasks are executed 'simultaneously'.

Tasks are used to monitor and control different jobs that occur simultaneously. Defining each job independently in a separate task does this.

Data are exchanged with other tasks and 'the world outside' via global variables.

Unlike procedure, tasks are not called since they are always present in a task queue and

A Process-Pascal program is divided into a heading and a body. The body is called a block. The heading gives the program name, object-type to be used in VIGO and the capabilities for the Controller (also used in VIGO). The block consists of seven sections: LABEL-, CONST-, TYPE-, VAR-, PROCEDURES and FUNCTIONS- and TASK declaration, where any except the last may be empty.

All that is defined before tasks is called the global section and you can have as many declaration sections as you want, in any order you want, including procedures and functions declarations. But, as in standard Pascal, things must be defined before they are used otherwise a compile-time error will occur.

Task, procedure and function declaration has a structure similar to a program; i.e. consists of a heading and a block. The symbols in the heading are different (TASK, PROCEDURE, FUNCTION instead of PROGRAM) and they end with a semicolon instead of a period. They can have their own constants, data types, and variables, even their own procedures and functions.

Tasks are different from procedures and functions at various points:

1. TASKs have their own memory area allocated for variables defined in a VAR section in the block. Termination of a task does NOT release this allocated storage.
2. TASKs have their own program counter and stack pointer and operate entirely autonomously from other tasks.
3. TASKs can not be nested.
4. TASKs are not called from a statement to execute.

The next chapters should be read in order to understand the proper use of task.

2.1 Task, an Introduction

Multitasking is a facility in Process-Pascal, which makes it possible to execute several sub-programmes simultaneously in the very same computer. These sub-programmes are called TASKs and are fundamental to Process-Pascal. They make it very easy to split up a program into manageable proportions where each TASK performs a distinct function.

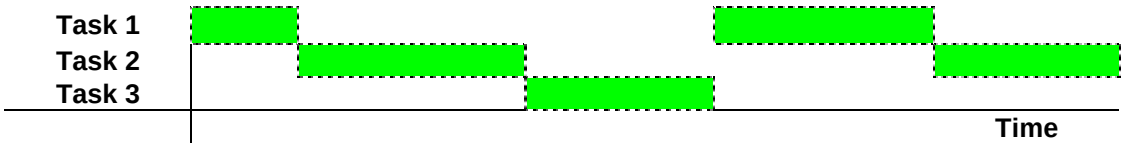
Multitasking is very useful for process control where the process can be controlled in real time.

A TASK is a section of code, which controls a part of the process, e.g. monitoring the keyboard for user input or controlling the valves on a blending unit etc. Each TASK will run and perform as much of its function as it wants to before it relinquishes control of the processor and lets another TASK run. While in reality the TASKs are not performed in parallel, the switching between them is done fast enough to make this a useful aid in visualising a system in real time. Switching from one TASK to another can be done in all parts of the program, including procedures. It can advantageously be used each time a delay appears or the TASK is waiting for some actions to take place, e.g. a certain level on an input signal or a TIMER to run out.

Switching to another TASK in such situations makes the program more efficient and waiting wastes no time.

The statement CHANGETASK does switching from one TASK to another, which is a standard procedure in Process-Pascal. The actual TASK calling CHANGETASK stops program execution in the TASK and relinquishes control of the processor to the following TASK in which the program execution continues from where it was last interrupted (e.g. by CHANGETASK).

The principle diagram below shows how the program execution is switching between a number of cyclic TASKs



2.2 Task types.

Process-Pascal handles 3 different types of TASKs: CYCLIC TASK, TIMEDINTERRUPT TASK and SOFTWIREINTERRUPT TASK. All 3 types of TASK can be used within the same program.

Cyclic TASKs are executed in sequence, where CHANGETASK switches to the following one in the sequence. The sequence is defined by the order of the TASKs in the program.

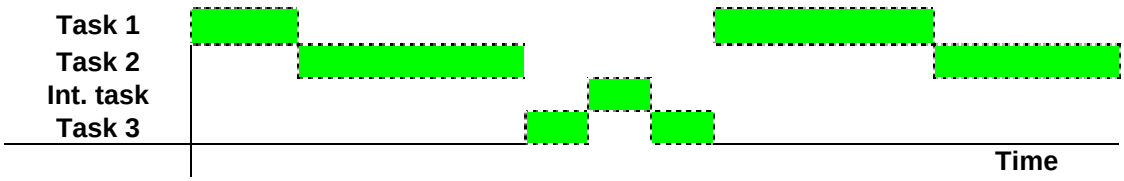
TIMEDINTERRUPT TASKs are executed at certain time intervals, as controlled by the programmer. The time periods are declared in seconds and the resolution is 1/128 second.

SOFTWIREINTERRUPT TASKs are executed each time a certain defined event occurs, e.g. the keyboard is activated and a TASK starts running to read which key was pressed and to undertake the appropriate action.

When a cyclic TASK is running and a timedinterrupt or softwireinterrupt TASK is ready to run, a CHANGETASK is forced in the cyclic TASK and control is given to the interrupting TASK. When the interrupting TASK has finished, i.e. reaches a CHANGETASK statement this CHANGETASK makes the earlier cyclic TASK continue where it was interrupted.

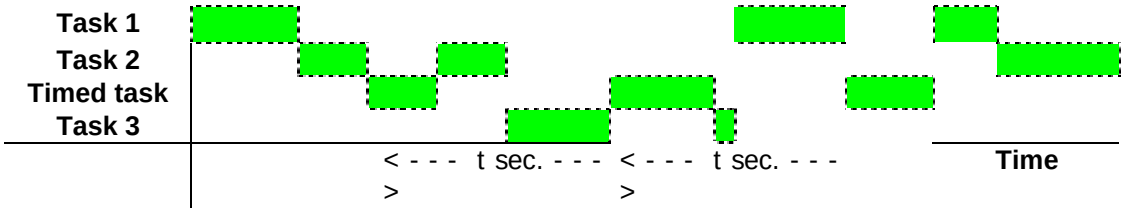
A timedinterrupt or softwireinterrupt TASK can not be interrupted by other TASKs.

The principle diagram below shows how the program execution is switching when a SOFTWIREINTERRUPT TASK interrupts a number of cyclic TASKs



The principle diagram below shows how the program execution is switching when a

TIMEDINTERRUPT TASK is interrupting a number of cyclic TASKs. The TIMEDINTERRUPT TASK is timed to t seconds.



2.3 How to Split a Program into Manageable Tasks.

Several separate TASKs may be monitoring and controlling various devices simultaneously. A single TASK can be written to monitor the keyboard for user input; another TASK is responsible for what is displayed on the screen; yet another TASK may be monitoring a flow meter waiting for a flow start. One TASK may be affected by another one, i.e. the keyboard TASK notes a key press which indicates the beginning of a process, so the TASK monitoring a flow meter indicates a start to take notice of a flow rate, which in turn causes the flow rate to be displayed on the screen by the display TASK.

Splitting up a program in TASKs is done by considering what tasks need to be performed simultaneously. Each of these tasks may then be implemented as a TASK itself. Taking similar examples as above, it can be seen that the tree task mentioned, monitoring the keyboard, monitoring a flow meter and displaying the data on the screen all need to be done simultaneously, and are therefore candidates of separate TASKs.

Proper *changetask* usage
A proper switching between tasks is an inherit part of the correct and adequate PROCESS-PASCAL application. The main concept behind it is that the ***changetask*** statement must be used in any part of the task where relatively long processor usage can be expected, for example loops.

Differences between interrupt tasks and event handling procedures
It is very important to understand a difference between multi-tasking programming and event-driven programming.
It is fact that event-driven programming plays a key role in MS Windows application development. In this case a relevant procedure (function) is called when a certain event occurs. It is performed and then the control is given back to the main application.
Multi-tasking programming is based on another concept. When a cyclic task is running and a timed interrupt or softwire interrupt task is ready to run, a ***changetask*** is forced in the cyclic task and control is given to the interrupting task. When the interrupting task has finished, i.e. reaches a ***changetask*** statement; this ***changetask*** makes the earlier cyclic task continue where it was interrupted. It means that timed interrupt or softwire interrupt tasks should always contain a LOOP statement and at least one ***changetask***. Otherwise the task will get a SUSPENDED status after the first run (if the LOOP is absent) or other tasks will never get back the control (if the ***changetask*** is absent).

3 Defining Data.

3.1 Variables

A variable is set off by three characteristics:

1: a name
2: a type
3: a current value

The variable is identified by a name. This name is used in the entire program.

When you declare a variable, you must state its type. A variable's type circumscribes the set of values, which it can possess, and the operations that can be performed on it.

The value of variables may change during program execution. When a variable has been declared, but before a value has been assigned to it, the variable is said to be undefined.

3.2 Identifiers

Any names denoting constants, types, variables, bounds, procedures, functions etc., are called identifiers.

They must begin with a letter, which may be followed by any combination and number of letters and digits. Corresponding upper-case and lower-case letters are considered equivalent. Letters can be in the range from 'a' to 'z', an underscore '_' and the Danish letters 'æ', 'ø' and 'å'.

Examples of identifiers:

Temperature MultiFunc ProcessTime FirstKey
ModePort1 This_Is_A_Very_Long_Identifier

Certain identifiers are reserved (word-symbols or reserved words). A reserved word must not be used as an identifier.

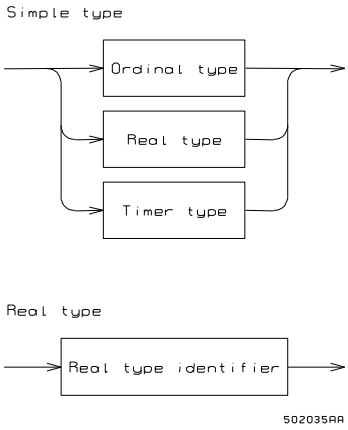
Process-Pascal provides a number of pre-declared identifiers. These pre-declared identifiers are not reserved words, but names for standard procedures, functions and so on. These names should not be used either, to avoid any mistakes. A complete list of all reserved words and pre-declared identifiers in Process-Pascal is given in chapter 35.3.

4 Simple Data Types

A program uses data of various formats and for various functions. The formats, and partly the function, are determined by the data type.

A data type defines the set of values a variable may assume and the basic operations, which may be applied to it. Every variable occurring in a program must be associated with one and only one type.

Simple data types define ordered sets of values and is one of the predefined types 'REAL', 'LONGREAL', 'TIMER' or an ordinal type.



4.1 Ordinal types

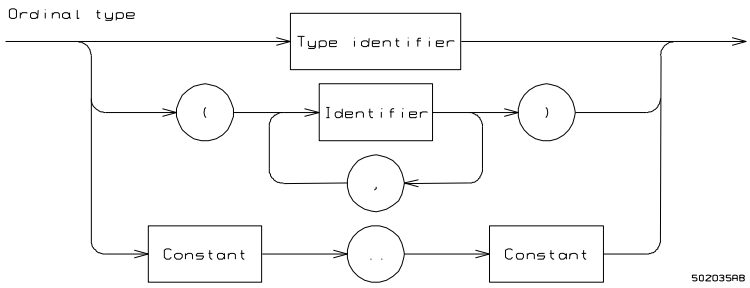
Ordinal types are a subset of simple types. Ordinal types are set off by four characteristics:

1: All possible values of a given ordinal type are an ordered set, and each possible value is associated with an ordinality, which is an integral value. Except for type integer values, the first value of every ordinal type has ordinality 0, the next has ordinality 1, and so on for each value in that ordinal type. A type integer value's ordinality is the value itself. In any ordinal type, each value other than the first has a predecessor, and each value other than the last has a successor based on the ordering of the type.

2: The standard function **Ord** can be applied to any ordinal type value to return the ordinality of the value.

3: The standard function **Pred** can be applied to any ordinal type value to return the predecessor of the value. The predecessor is defined by Pred(x) < x and Ord(Pred(x)) = Ord(x) - 1.

4: The standard function **Succ** can be applied to any ordinal type value to return the successor of the value. The successor is defined by Succ(x) > x and Ord(Succ(x)) = Ord(x) + 1.



Process-Pascal has 6 predefined ordinal types: **Integer, Byte, Word, Longinteger, Boolean,** and **Char**. In addition, there are two other classes of user-defined ordinal types: enumerated types and sub-range types. These types are described in the USERDEFINED TYPES chapter.

4.2 The type BOOLEAN.

A boolean value is one of the logical truth values denoted by the predefined constant identifiers **false** and **true**. In Process-Pascal, the predefined constant identifier **Off** equals **false** and the predefined constant identifier **On** equals **true**.

Relational operators (=, <>, <=, <, >, >=) can be used on a boolean and the following relationships holds:

```
False < True
Ord(False) = 0
Ord(On) = 1
False = Off
True = On
```

Pre-declared BOOLEAN functions, i.e., pre-declared functions, which yield a BOOLEAN result, are:

```
BufferEmpty(buf)true if the buffer is empty
BufferFull(buf)  true if the buffer is full
Odd(i)           true if the integer i is odd
```

The buffer functions are described in details in the BUFFER chapter.

4.3 The type CHAR

This type's set of values is characters, ordered according to the ASCII character set. The function call **Ord(ch)**, where ch is a char value, returns ch's ordinality, which means the ASCII value for the character.

Any value of the type char can be generated with the standard function **Chr(value)**.

A character enclosed in apostrophes (single quotes or double quotes) denotes a value of the char type.

To represent a single quote, enclose it in double quotes. To represent a double quote, enclose it in single quotes.

Examples of the char type:

'a' 'H' '8' "e" """" ""

4.4 The type INTEGER.

There are four predefined integer types in Process-Pascal: **integer, byte, word** and **longinteger**. Each type denotes a specific subset of the whole numbers, according to the following table:

TYPE	RANGE	FORMAT
byte	0 .. 255	unsigned 8-bit
word	0 .. 65535	unsigned 16-bit
integer	-32768 .. 32767	signed 16-bit
longinteger	-2147483648 .. 2147483647	signed 32-bit

Arithmetic operation with type integer operand use 8-bit, 16-bit and 32-bit precision, according to the following rules:

1: The type of an integer constant is the predefined integer type with the smallest range that includes the values of the integer constant.

2: Binary operations can be performed with all integer types. For a binary operator, both operands are converted to their common type before the operation. The common type for a byte and a word is word, which means that a binary operation on a byte and a word converts the byte to a word and then the operation is performed.

3: The expression on the right side of an assignment statement is evaluated dependently from the type of the variable in the expression and the type on the left side.

A type integer is converted to another integer type through typecasting. Typecasting is automatically performed during compilation.

A special typecasting can be performed for integer types to boolean array types and visa versa through a CONVERT function. The CONVERT function performs the typecasting according to the following table:

INTEGER TYPE	BOOLEAN ARRAY SIZE
byte	8
integer	16
word	16
longinteger	32

Examples of using the CONVERT function:

```
TYPE
        BIT8  = ARRAY[0..7]  OF BOOLEAN;
        BIT16 = ARRAY[0..15] OF BOOLEAN;
        BIT32 = ARRAY[0..31] OF BOOLEAN;

VAR
        Bit8Var  : BIT8;
        Bit16Var : BIT16;
        Bit32Var : BIT32;
        ByteVar  : BYTE;
        IntVar   : INTEGER;
        LIntVar  : LONGINTEGER;

BEGIN
        ByteVar:=Convert(Bit8Var);      (* convert an 8 bit boolean array to a byte *)
        Bit16Var:=Convert(IntVar);      (* convert an integer to a 16 bit boolean array *)
        Bit32Var:=Convert(LIntVar);     (* convert a longinteger to a 32 bit boolean array
                                           *)
```

This CONVERT function is very useful when you want to mask out some bits, or to read a combination of bits as data in conjunction with digital input and output.
NOTE: the boolean array must start with index 0.

4.5 The type REAL

A real type has a set of values that is a subset of real numbers, which can be represented, in floating-point notation with a fixed number of digits.

There are two kinds of real types: **real** and **longreal**.

The type **real** occupies 4 bytes of memory with a format according to the IEEE 754 standard for short real format (same format used in the 80x87 math processor for single type real), providing a range of 3.4 * 10E-38 to 3.4 * 10E38 with 7 significant digits.

The type **longreal** occupies 8 bytes of memory with a format according to the IEEE 754 standard for long real format (same format used in the 80x87 math processor for double type real), providing a range of 1.7 * 10E-308 to 1.7 * 10E308 with 15 significant digits.

4.6 The type TIMER

The type **timer** occupies 4 bytes of memory and is assigned as a **real**. The value for a variable of timer type is in seconds. A variable of type timer counts down with a resolution of 1/128 second. The countdown **continues** through negative values.

The timer stops counting down when the power is off.

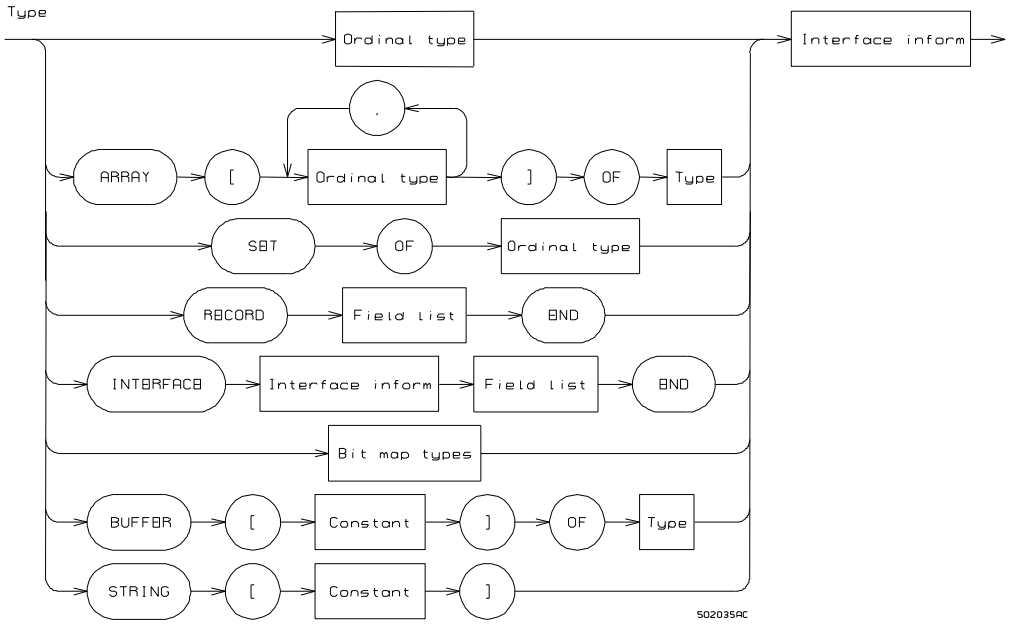
A timer type variable can be used anywhere in the program. It is commonly used by assigning a value to it and afterwards testing if the value of the variable is <= 0.0.

The number of defined variables of timer type has no affect on the program execution time. TIMERS can not be set to values higher than 1.6777 * 10E7 corresponding to 4660 hours or 194 days.

5 Structured Types

Process-Pascal provides facilities for creating collections of data types in the form of structured types. Although data types can be quite sophisticated, each must be ultimately built from unstructured simple types.

A structured type, characterised by it's structuring method and by its component type(s), holds more than one value. If a component type is structured, the resulting structured type has more than one level of structuring. A structured type can have unlimited levels of structuring.



Each of the methods for structuring types is described in separate chapters.

6 Variable Declaration

Every variable identifier occurring in a program must be introduced in a variable declaration. This declaration must textually be introduced before any use of the variable.

A variable declaration introduces a variable identifier and its associated data type by listing the identifier followed by the type. The type given for the variables can be a type identifier previously declared in a type declaration part in the same block, in an enclosing block or it can be a new type definition. The reserved word **VAR** heads the variable declaration part. It is allowed to type **VAR** more than one time in the same variable declaration part.

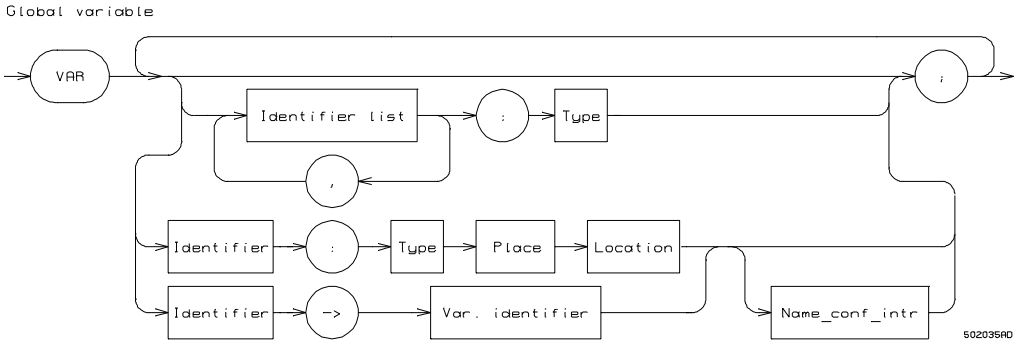
Variables can be declared to reside inside the controller or externally in other modules at a net-address. The compiler can allocate variables, or they can be declared to reside at specific memory addresses for special applications.

Variables declared before tasks, and outside procedures and functions are called **global variables** and reside in a global data section. Variables declared within a task, but outside procedures and functions are called **local variables** and reside in a local data section for the specific task. Variables declared within procedures and functions are also called local variables, but these variables are only known within the procedure or function in which they are defined.

6.1 Global variables

All the global identifiers used in a Process-Pascal program are converted to a number by the compiler. These softwire numbers are used as an entry key to the softwire list which contains structured information on each individual global variable and constant, used in the particular program.

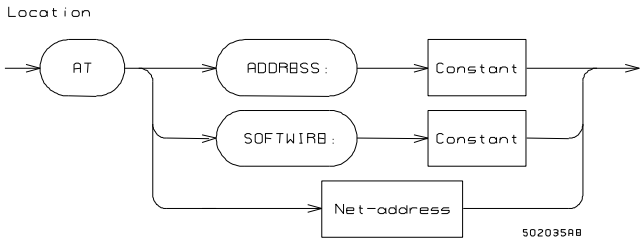
Variables of the same type can be declared by a list of identifiers, separated by a comma, followed by a colon and the type of the variables.



Examples of variable declarations:

```
VAR
    LineNo, PageNo : INTEGER;                (* These variables *)
    Color : BYTE;                            (* are allocated   *)
    Process_On, AlarmState : BOOLEAN;        (* by the compiler *)
    Wait, LightTime : TIMER;
    Limit : REAL;
```

Variables can be defined to reside at a specific address in memory, at a specific softwire number or at a net-address. If a variable is declared to reside at a softwire number or at a net-address, memory has already been allocated for it.



The **address** clause is followed by an absolute address in memory and the identifier specifies only one identifier.

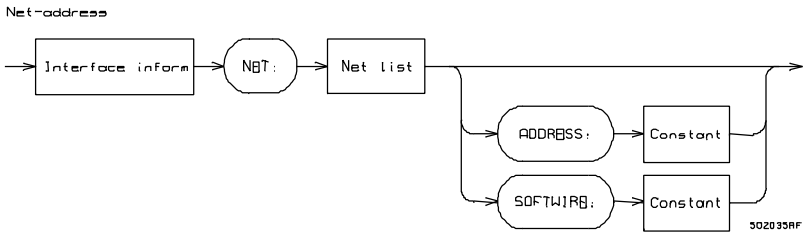
Example of a variable declaration to a specific memory address:

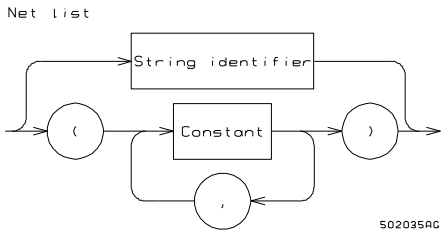
```
VAR
    LightValue : WORD AT ADDRESS : $00FFFF08;
```

The **softwire** clause is followed by a number from the softwire table. The declaration is rarely used without a net-address because a global variable must be declared to generate an entry in the softwire table.

6.2 Variables at P-NET

Variables that are physically located in a module that is connected to P-NET must be declared to reside at a certain location, defined by a net-address. When variables are declared to with a net address, no memory space is allocated in the controller.





The net-address denotes a net list, followed by an address, which can be an absolute address or a softwire number.

The **net list** holds an ordered set of numbers, which describes the path to the device, i.e. denoting the port-numbers and P-NET numbers for the module containing the variable.

The net list can also be a string-identifier. This means that the net list can be a string-variable and the P-NET node address for the module can therefore be set or changed during the program execution.

Example of a variable declaration, using a net list:

```
VAR
    DigModule : PD3221 AT NET: ( 1,64);
```

This variable declaration defines an entire interface module of the type PD3221 with all channels and registers to reside at P-NET. The device is connected to the Controller via P-NET at port 1, and the device node address is 64.

DigModule is a global identifier for the entire interface module and can be used as any other identifier throughout the program.

PD3221 is the type of the variable and is a pre-declared type specifying the internal organisation of the channels within the module.

AT NET specifies that the declared variable is an external variable that is located on P-NET. Any access to that variable is performed via the network. The following parameters (1, 64) specify where the module is located, as seen from the controller. The first parameter indicates the communication port (Port 1 in this case), and the next parameter defines that the module is expected to have node address number 64.

The address and softwire clause denotes a specific address or a SoftWire entry in the module defined by net list.

Example of a variable declaration, using a net list and an address clause:

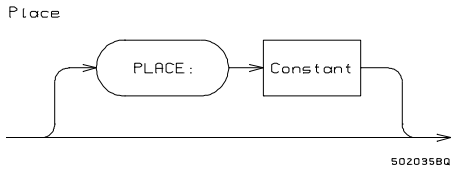
```
VAR
    Mixer1 : MixerController AT NET: ( 1,37) ADDRESS: $0C00;
```

Examples of variable declarations, using a net list and a SOFTWIRE clause:

```
VAR
    BeltControl : BeltConType AT NET: ( 1,38) SOFTWIRE: $92;
    ExtInt : Integer AT NET: (2,3) SOFTWIRE: $124;
```

If you want to declare a variable to reside on a fixed SoftWire number, e.g. a global

database for a number of controllers, you must use the PLACE clause (examples of this can be found in the system files for the Controllers).

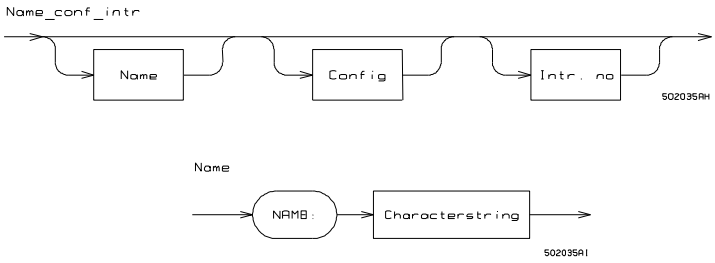


Example on a variable declaration, using the PLACE clause:

```
VAR
    DataBase : ARRAY[1.2000] OF INTEGER PLACE: 200;
```

This declaration will fix the variable at softwire no. 200. You must assure that the declaration is made at a location before the compiler would have generated the softwire number automatically, i.e. it is to late to place a variable at softwire number 200 if you alredy have declared 300 variables.

A variable can be declared with a **Name**, which declares a name as a stringconstant for the variable. This name can be used as a string when an error occurs for the variable. See details about errors and error handling in the WHEN ERROR chapter.



Examples of variable declarations, using a name:

```
VAR
    DigModule : PD3221 AT NET: ( 1,35)
                    NAME : 'Digital module panel 1';
    AnaModule : PD3240 AT NET: ( 1,38)
                    NAME : 'Analog controlunit 22';
```

When using NAME on variables of interface type (modules) conforming to the section INTERFACE DECLARATION in the P-NET standard, each channel can get it's own name. NAME for the module belongs to channel 0, the service channel. When using NAME on other variables than interface modules, each variable can get only one name.

6.3 Config

A **CONFIG** clause can be used to assign a value to the variable when calling a CONFIG statement in the program. This is done by typing the following

CONFIG: Procedureidentifier

after the type and name for the variable. The procedure will be executed when a CONFIG statement is executed.

See also the examples for the Service and Config programmes in the examples directories in the Process-Pascal library.

Examples of variable declarations, using CONFIG:

```
VAR
DigModule : PD3221 AT NET: ( 1,35)
            NAME : 'Module at CIP unit'
            CONFIG : SetByte(.Service.ModuleConfig, WatchDog);

AnaModule : PD3240 AT NET: ( 1,38)
            NAME : 'Inlet control unit'
            CONFIG : Standard_PT100(.Analog_In_4);
```

The procedure call passes the variable itself as a default parameter. When the variable is of complex type, a part of the variable can be selected as the first parameter. When the variable is an entire module, a channel or even a register can be selected to be the parameter, see the example above. The resulting procedure calls for the above Config clause will be:

```
            SETBYTE(DIGMODULE.SERVICE.MODULECONFIG, WATCHDOG);
            STANDARD_PT100(ANAMODULE.ANALOG_IN_4);
```

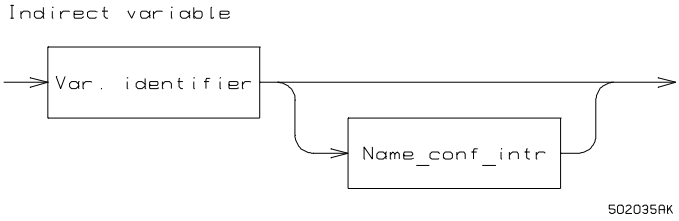
These procedure calls can be seen in the LIST-file, placed instead of the CONFIG statement. The Config procedures are declared in the files called Config4.inc for PD 4000 and in Config5.inc for PD 5000.

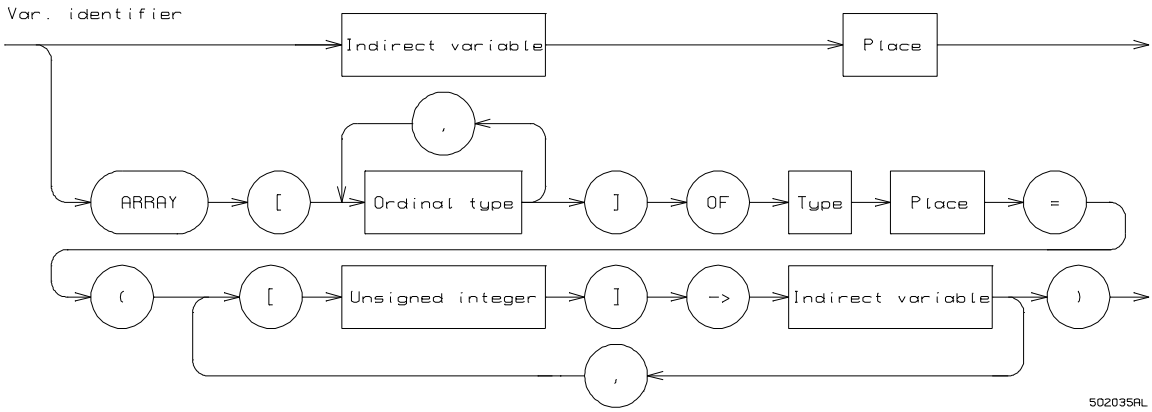
6.4 Indirect variables

The previous declarations shows how to declare an entire interface module. But when writing a program it is often more convenient with a more detailed specification of inputs and outputs.

Variables can be declared indirectly, which means that a variable is declared to reside at the same location (the same memory address) as a previous declared variable, but is accessed with a different identifier. An indirect variable can be a sub-variable of a previous declared variable.

Indirect variables are declared by an identifier followed by -> and followed by an identifier for a previously declared variable. This previously declared variable can be of any type. When using this way for variable declaration, a new identifier is declared with the same type and the same address as the variable on the right side of the -> sign.





This way of declaring variables is not a part of standard Pascal.

The **indirect variable** is a variable reference, and can be a whole structured variable, a specific component of a structured variable or a variable of simple type.

The following example demonstrates how the indirect variable declaration is done in practise.

```
VAR
  UPI : PD3221 AT NET: (1,64);     (* Defines a UPI slave module *)

  AgitatorCh -> UPI.Digital_IO_4;  (* Defines the Digital_IO channel No 4 *)
  OverfillCh -> UPI.Digital_IO_6;  (* Defines the Digital_IO channel No 6 *)

  Agitator -> AgitatorCh.FlagReg[7];      (* Defines an Out Flag  of the Digital_ IO channel *)
  Overfill -> OverfillCh.FlagReg[6]; (*  Defines an In Flag  of the Digital_ IO channel   *)

  TempCh -> UPI.Analog_In_1;       (* Defines the 1st Analog Input channel *)
  Temp -> TempCh.AnalogIn;         (* Defines the variable containing a value of the
analog
                                      input No 1 *)
```

Then, these variables can be handled as ordinary variables. For example the following statements will all start the agitator:

```
UPI.Digital_IO_4.FlagReg[7] := true;
AgitatorCh.FlagReg[7] := true;
Agitator := true;
```

And the following statement will call the *AlarmProc* procedure in the case of the level detector activation:

```
IF Overfill THEN AlarmProc;
```

Indirect arrays can be used to assemble particular variables, or part of variables, in a structured manner. This can be used to make easier and more understandable programmes.

The next example demonstrates this powerful feature.

```
VAR
        DigModule1 : PD3221 AT NET (2,51);
        DigModule2 : PD3221 AT NET (2,52);

        Valves -> ARRAY[1..MaxNumberOfValves] OF DigitalCh =
                ([1] -> DigModule1.Digital_IO_1,
                 [2] -> DigModule1.Digital_IO_2,
                 [3] -> DigModule1.Digital_IO_3,
                 [4] -> DigModule1.Digital_IO_4,
                 [5] -> DigModule2.Digital_IO_1,
                 [6] -> DigModule2.Digital_IO_2,
                 [7] -> DigModule2.Digital_IO_3,
                 [8] -> DigModule2.Digital_IO_4);
```

To access an IO channel in either of the two digital modules, i.e. a valve, an indirect element in the variable VALVES is accessed:

```
        Valves[ValveNumber].FlagReg[7]:=ON;
        IF Valves[3].Counter <= 20 THEN
```

Examples of indirect variable declarations using the NAME clause:

```
VAR
    Start->DigModule.Digital_IO_1
        NAME :'Start button for production';
    WaterTemp->AnaModule.Analog_In_1.AnalogIn
        NAME :'Water temperature';
```

The name 'Start button for production' is connected to the variable Start, which means that the name can be used as a string when an error occurs in accessing channel 1 in DigModule. See the WHEN ERROR chapter how to use and retrieve the declared NAME.

The CONFIG clause can also be used on indirect variables.

Examples of indirect variable declarations using the NAME and CONFIG clause:

```
VAR
    Start->DigModule.Digital_IO_1
                NAME :'Start button for production'
                CONFIG: DigitalInput;
    WaterTemp->AnaModule.Analog_In_1
                NAME :'Water temperature'
                CONFIG: Standard_Pt100;
```

7 Pointer Types

All the previously discussed data types have the ability to hold data. A POINTER holds a different type of information, - the location where data are stored. Process-Pascal provides the use of pointers, as static variables, which means that the pointer variables are declared in the program and following denoted by their identifiers and they exist in the entire execution of a block (program, task, procedure or function). Pointer types can not be allocated dynamically during program execution.

A pointer is always specific to an other data type and it can only point to a previous declared variable of that type or it can point to NIL. If a pointer is not initialised or pointing to NIL, the value of the pointer is undefined and an error code is generated (Error3 = $18). The standard function PointerOK can be used to test if a pointer is valid.

A pointer holds information of a variables softwire number and an offset, and occupies 12 bytes of memory.

Examples of pointer types:

```
TYPE
        RealPointer = POINTER TO REAL;

VAR
        Weight -> WeightModule.Ch1.Flow;
        Flow   -> FlowMeter.Flow;
        MeasuredValue : RealPointer;

BEGIN
  IF MeasuringModule = FlowModule THEN
     MeasuredValue -> Flow       (* set pointer to Flow register in flowmeter *)
  ELSE
     MeasuredValue -> Weight;   (* set pointer to Flow register in weight module *)

     Display(MeasuredValue:6:1);        (* display flow from either flowmeter or weight module
                                        as
                                        measured value *)

     IF MeasuredValue > MaxFlow THEN ReduceFlow;
                                (* compare MaxFlow to the value that MeasuredValue is
                                pointing to *)
```

The pointer itself must be declared to reside internally, but it is allowed to point to as well internal as external variables.

A pointer type may be a part of another type, e.g. as a field in a record.

```
        MyRecordType = Record
                ASimpleVariable:        Integer;
                PointerVaraible:        POINTER TO REAL;
              END;
```

8 Constants

The values 5, 1.25, -357 and TRUE in a program are called constants. A 5 in the program can only take the value 5, so 5 is a constant value. A constant can not change value during program execution.

A constant definition introduces an identifier as a synonym for a constant. The reserved word CONST heads the constant definition part. Constant values can be a number, a constant identifier, a character, a string or a structured constant (see the STRUCTURED CONSTANT chapter).

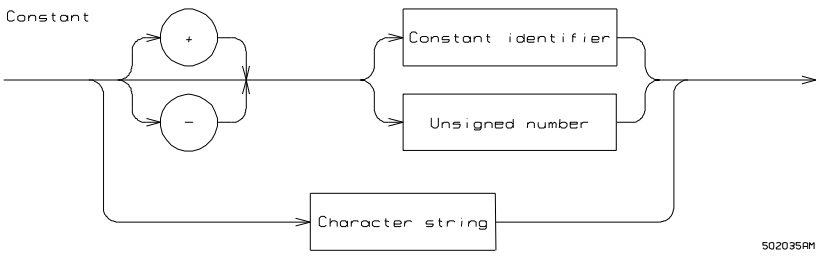
The use of constant identifiers generally makes a program more readable and acts as a convenient documentation aid. It also allows you to group machine-dependent quantities at the beginning of the program where they can be easily changed. You only have to change the value of a constant in the CONSTANT declaration part, instead of changing the constant value in all parts of the program where it is used.

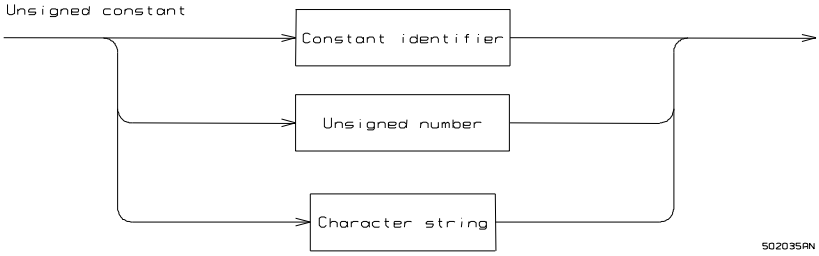
Examples of constant declarations:

```
CONST
    Max_Valves = 100;
    CursorStepX = 6;
    PageSize = 50;
    Blank = '          ';
    Manual_Set = '1.0';
    WaitTime = 2.7;
    AlarmOn = TRUE;
    CrLf = #13#10;
```

The compiler determines the type for the constant, depending on the syntax and range. However, the constant can be forced to get a specific type by using a type identifier in the declaration. E.g.:

```
    PD340Type = WORD(56);
```





9 Comments

The readability of a Process-Pascal program can be improved by inserting blanks, blank lines and notes in it. The notes can be inserted to remind you (or someone else who reads or maintain your program) what certain variables mean, what certain functions or procedures do, and so on. These notes are known as COMMENTS.

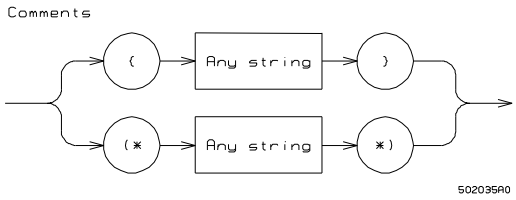
A program may contain as many comments as you want and a comment may contain any sequence of characters.

A comment begins with a left curly brace { or a left parenthesis and an asterisk, (*, and ends with a matching right curly brace } or a matching asterisk and a right parenthesis, *). A comment that contains a dollar sign immediately after the opening { or (* is a compiler directive. See chapter 35.5.

You can start a comment with a left curly brace {, which signals to the compiler to ignore everything until after it sees the right curly brace }. This allows for a limited form of comment nesting, because a comment beginning with a { ignores all (* and visa versa.

Example of a comment:

```
a:=7; (* This is a comment for the statement *)
```



It is suggested to use one type of comment markers for program comments and compiler directives and an other type for temporary program parts. This is very useful during program development and makes it easy to use comment nesting.

10 Expressions and Assignments

10.1 Expressions

An expression is a rule for calculating a value based on the observation of conventional rules of algebra for left-to-right evaluation of operators and operands.

The value that is calculated depends on the value of the constants and variables, which are included in the expression, and on the operators and functions that are used in the expression.

10.2 Operators

Expressions apply the normal arithmetic operators, logical operators and relational operators.

10.3 Arithmetic operators

The arithmetic operators are: +, -, *, and /, where * is multiplication and / is division.

These operators can be used on integer types, real types and timer types. The result type for these operations depends on the value type that is calculated. This is caused by the automatic typecasting during compilation.

Examples of expressions with arithmetic operators:

```
x + y
51.8 - 2
arc * number
10 / 2.45
```

Furthermore are two operators which only operates on integer operands. These are DIV and MOD.

The DIV operator performs an integer division (i.e. value is not rounded).

Examples of the DIV operator:

expression	result
15 **DIV** 6	2
15 **DIV** 7	2
-15 **DIV** 5	-3

The MOD operator returns the remainder obtained by dividing its two operands.

Examples of the MOD operator:

expression	result
15 **MOD** 6	3
-15 **MOD** 7	-1
15 **MOD** 5	0

10.4 Logical operators

The logical operators are NOT, AND and OR. The logical operators can operate on all integer types and on BOOLEAN types.

The NOT operator performs a bitwise negation on one operand.

Examples of the NOT operator:

operand type	expression	result
byte	**NOT** $00	$FF
word	**NOT** $0101	$FEFE
boolean	**NOT** TRUE	FALSE
boolean	**NOT** FALSE	TRUE

The AND operator performs a bitwise and on the operands.

Examples of the AND operator:

operand types	expression	result
byte	$55 **AND** $11	$11
word	$0202 **AND** $0101	$0000
boolean	TRUE **AND** TRUE	TRUE
boolean	TRUE **AND** FALSE	FALSE

The OR operator performs a bitwise or on the one operands.

Examples of the OR operator:

operand types	expression	result
byte	$55 **OR** $11	$55
word	$0202 **OR** $0101	$0303
boolean	FALSE **OR** TRUE	TRUE

10.5 Relational operators

The relational operators are =, <>, >, <, >= ,<= and IN.

The relational operators can be used on all simple data types: boolean, byte, char, integer, longinteger, longreal, real and timer. Different types can be compared, because of the automatic typecasting. Furthermore strings can be compared according to the ordering of the extended ASCII character set. The IN operator is used to test for membership of a SET type operand. The result type is always a boolean, i.e. true or false.

Examples of relational operators:

```
WaitTime <= TimeOut
Weight > SetPoint
PassWord <> PassCode
InputChar IN Digits
```

10.6 String operator

Process-Pascal allows the + operator to be used to append two string operands. The result of the operation StrA + StrB, where StrA and StrB are of string types, is the addition of the strings with the first character from StrB positioned after the last character from StrA and the length is the integer addition of the two string lengths. If the resulting string is longer than the result type, the string is truncated to the max string length of the result type.

The value of expressions can be converted to strings by adding a size-specifier and a format-specifier to the expressions that you want to convert. The syntax is as follows:

```
Str := expression : size-specifier : format-specifier
```

The size-specifier denotes the number of characters that is converted to characters for the expression. The format-specifier is a value for how to convert the expression to the string.
If result type for the expression is of type TIMER, REAL or LONGREAL, format has the following meaning:

0-..	Number of digits to the right of the decimal point.
-1	The variable is represented with floating-point.
-2	The variable is represented with exponent. For the type TIMER or REAL the exponent is always 2 digits and a sign. For the type LONGREAL the exponent is always 3 digits and a sign.

If the expression is a simple type different from TIMER, REAL or LONGREAL, format has the following meaning:

0	Decimal representation with leading spaces.
-3	Hexadecimal representation.
-4	Binary representation.
-5	Decimal representation with leading zeros.

If the expression contains operators, it must be enclosed in brackets.

Example:

```
(* r is a real with the value 25.61 and str is a string[35] *)
Str := 'The value of r is : ' + r:5:2 ;

After this operation Str holds the following characters:

        The value of r is : 25.61

Str:='The weight is : ' + (Weight / 1000.0):6:1 + 'T';
```

Weight is assumed to be a variable that holds the value for a weight in kg.

10.7 Operator precedence

The operators are classified in 5 categories ordered by their precedence, the first with the highest precedence.

The table below shows the rules of operator precedence and should be referred to whenever you are in doubt of the exact rules.

1	Unary minus	(minus with only one operand).
2	NOT operation	(boolean negation).
3	Multiplying operators	(*, /, DIV, MOD, AND).
4	Adding operators	(+, -, OR).
5	Relational operators	(=, <>, >, >, <=, >= ,IN).

11 Statements

A program must do something with its data and its input/output. What exactly the program is doing, is described in statements. Statements describe algorithmic actions that can be executed. Statements are either simple or structured. Please refer to the STATEMENT syntax diagram in the chapter SYNTAX DIAGRAMS.

11.1 Simple statements

A simple statement is a statement that doesn't contain any other statements. Simple statements are assignment statements, procedure statements or the empty statement. The empty statement consists of no symbols and denotes no action.

11.2 Assignment

The most fundamental of statements is the assignment statement. It specifies that a newly computed value be assigned to a variable. The value is specified by an expression. The variable may be a simple variable or an entire structured variable, located in the computer or in a module on the P-NET. The assignment statement has the following form:

```
identifier := expression
```

where the identifier is a variable identifier. The assignment statement is a simple statement.

Examples of the assignment statement:

```
SetPoint:= Recipe[i].Parts / 100 * Scale
DrainValve:= ON
DigitalModule.Ch20.FlagReg[7]:= OFF
Weight_Timer:= 10.0
```

11.3 Procedure statement

Another simple statement is the procedure statement, which activates the named procedure which is a subprogram specifying another set of actions to be performed on data.

Examples of procedure statements:

```
PrintOut
Picture_11(No-Scroll)
StopMixing(MixerNo, StopCommand)
```

See the chapter "PROCEDURES AND FUNCTIONS" for more details about procedures.

11.4 Structured statements

Structured statements are constructs composed of other statements that are to be executed in sequence (compound statements and with statements), conditionally (conditional statements) or repeatedly (repetitive statements).

11.5 Compound statement (begin end)

The compound statement specifies that its component statements are to be executed in the same sequence as they are written. The symbols BEGIN and END act as statement brackets, and each statement are separated by semicolons. The semicolon is not a part of the statement, it is only used to separate them. An extra semicolon before an END does no harm because an empty statement is assumed between the semicolon and the END.

Example of a compound statement:

```
BEGIN
   SetPoint:=0;
   ErrorMessage:=FALSE;
   PrintOut;
END;
```

11.6 Conditional statement (if then else)

The if statement specifies that a statement be executed only if a certain condition is true. The condition is the result of a boolean expression which produces TRUE or FALSE. If the expression produces true, then the statement following the symbol THEN is executed. If the expression produces false and the ELSE part is present, then the statement following the symbol ELSE is executed. If the ELSE part is not present, no statement is executed.

Examples of IF statements:

```
IF Sec = 60 THEN Min := Min + 1;
IF Min = 60 THEN
BEGIN
   Hour := Hour + 1;
   Min := 0
END;

IF x > z THEN
    largest := x
ELSE
   largest := z;
```

Please note, there is never a semicolon after the boolean expression or before an ELSE, because semicolon is used to separate statements, not to end statements.

If more than one statement must be executed after the expression, the compound statement is necessary, see the second example above.

IF statements can be nested in as many levels as you want to. You should not to use too many levels, because it can be hard to avoid getting the different IF THEN and ELSE's mixed up.

11.7 Conditional statement (case)

The CASE statement consists of an expression (the selector) and a list of statements, each prefixed with one or more constants (called case constants) or with the symbol ELSE. The selector must be of any ordinal type (boolean, byte, char, word or integer), but longinteger, and the ordinal values of the upper and lower bounds of that type must be within the range -32768 to 32767. Each case constant must be associated with at most one of the statements.

The CASE statement executes the statement prefixed by a CASE constant equal to the value of the selector or a CASE range containing the value of the selector. If no such CASE constant of the CASE range exists and an ELSE part is present, the statement following the ELSE is executed. If there is no ELSE part, nothing is executed. The ordering of the case constants has no influence on the selection for execution.

The statement after the CASE constant can be a simple statement or a compound statement. When the statement has been executed, the program continues with the statement after the entire CASE statement.

Examples of CASE statement:

```
CASE Number OF
   1: Figure := 2;
   2: Figure := 45;
   3, 4, 5: Figure :=0;
   6..10: Figure := 100
END;

CASE Digit OF
   '1': BEGIN
           Value :=0;
           Score :=2
        END;
   '2': Value :=3;
   '3': BEGIN
           Value :=7;
           Score :=0;
           PrintOut
        END
   ELSE PrintOut
END;
```

11.8 While statement

A WHILE 'expression' DO statement contains an expression that controls the repeated execution of a statement, which can be a compound statement.

The expression that controls the repetition must be of type boolean. The execution of the statement after WHILE 'expression' DO is done zero or more times. Before the statement is executed the expression is evaluated. The statement is executed as long as the expression is true, otherwise the WHILE statement terminates. If the expression is false at the beginning, the statement is not executed at all.

Because the expression is evaluated for each iteration, you must be careful to keep the expression as simple as possible.

Examples of WHILE statement:

```
While BufferEmpty(KeyBoardBuffer) DO ChangeTask;

While TO1.AnalogIn > 35.0 DO
BEGIN
   FeedBackControl;
   ChangeTask
END;
```

11.9 Repeat statement

A REPEAT statement contains an expression that controls the repeated execution of a statement sequence within that repeat statement. The general form for the repeat statement is
REPEAT statement(s) UNTIL expression. Note that it is a sequence of statements that the repeat statement executes.

The expression that controls the repetition must be of type boolean. Opposite the WHILE statement, the execution of the statements after REPEAT is always done at least once. After each execution of the sequence of statements the boolean expression is evaluated. Repeated execution is continued until the expression becomes true.

Because the expression is evaluated after each iteration, you must be careful to keep the expression as simple as possible.

Examples of REPEAT statement:

```
REPEAT
   ChangeTask;
   Difference := SetPoint - TO1.AnalogIn;
UNTIL HeatControl = OFF;

REPEAT
   Number := Number + 1;
   LoopControl := LoopControl - 1
UNTIL LoopControl = 0;
```

Note that the second example performs correctly for LoopControl > 0 when entering the loop, but if it is less than zero, the loop will go forever.

11.10 For statement

The FOR statement indicates that a statement, which can be a compound statement, to be repeatedly executed while a progression of values is assigned to the control variable of the FOR statement.

The FOR statement has the form:

```
FOR controlvariable := initialvalue TO finalvalue DO statement.
```

The control variable must be of an integer type and declared in the same scope that the FOR statement appears. The initial value and the final value must be ordinal types compatible with the control variable. The initial value and the final value can be expressions. The initial value is evaluated only once and the final value is evaluated each time, before the statement contained by the FOR statement is executed.
The statement contained by the FOR statement is executed once for every value in the range initial value to final value. The control variable always starts off at initial value.

A FOR statement can use TO or DOWNTO for assigning values to the control variable. When a FOR statement uses TO, the value of the control variable is incremented by one for each repetition. If the initial value is greater than the final value, the contained statement is not executed. When a FOR statement uses DOWNTO, the value of the control variable is decremented by one for each repetition. If the initial value is less than the final value, the contained statement is not executed.

The control variable may be altered in the contained statement without causing an error.

The control variable is incremented/decremented when the contained statement has been executed. After the FOR statement is executed, the value of the control variable is undefined.

Examples of FOR statement:

```
FOR i:=1 TO NumberOfVAlves DO Valves[i].FlagReg[7]:=OFF;

FOR n:= Start TO Stop DO
BEGIN
   Recipe[n].Parts :=0;
   Recipe[n].Machine :=0
END;

FOR sl:= 50 DOWNTO 25 DO
    IF Data[sl].AlarmFlag THEN Data[sl].Counter:=0;
```

11.11 Loop statement

The LOOP statement specify that the contained statements are executed repeatedly forever and the loop can only be left by a WHEN ERROR statement (see the INTERRUPT chapter).

The LOOP statement has the following form:

```
LOOP statements END .
```

12 Array

For handling great amounts of data, it is often convenient to store these data in a structured way. An array structure is an example of a data structure where a group of data has been ordered in a certain pattern. An array is stored as a contiguous sequence of variables, all of the same type.

Arrays have a fixed number of components of one type, the component type. The component type follows the word **of** in the syntax for an array:

```
array_type : ARRAY[firstindex..lastindex] OF type
```

12.1 One dimensional arrays

The index type specify the number of elements. Valid index types are all ordinal types except longinteger and subranges of longinteger.

The index type must be a constant identifier or a constant. The index must not be a negative value.

Example of array declarations:

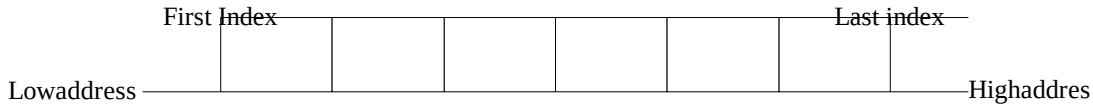
```
Data : ARRAY[1..MaxNumber] OF INTEGER;
SetPoints : ARRAY[FirstSetPoint..LastSetPoint] OF REAL;
```

An element in an array is referred with an index, where the index can be an expression. The result of the expression must be an ordinal type and the value should be within the specified index range. If the index value is less than the first index, then the first index is referred. If the index value is greater than the last index, then the last index is referred.
If the index value is out of range, an error is generated.

Examples of indexing an array:

Data[4]	denotes the fourth element in Data
Data[MaxNumber]	denotes the last element in Data

The component with the lowest index is stored at the lowest memory address as shown below:



The values of all elements in an array can be copied to a corresponding array by only one assignment.

Example:

```
VAR
   a,b :ARRAY[1..5] OF REAL;

BEGIN
   FOR i:=1 TO 5 DO a[i]:=0;    (* init array *)

   b:=a;                        (* copy array *)
```

12.2 Multidimensional arrays

Each element in an array can be an array and the index types specify the number of elements, one for each dimension of the array.

The array can be indexed in each dimension by all values of the corresponding index type, which means that the number of elements is the number of values in each index type. The number of dimensions is unlimited.

If an array type's component type is also an array, you can treat the result as an array of arrays or as a single multidimensional array. The following examples are interpreted the same way in the compiler:

```
ARRAY[1..100] OF ARRAY[1..5] OF REAL
ARRAY[1..100,1..5] OF REAL
```

An element in a multidimensional array is referred with a number of indexes, corresponding to the number of dimensions in the array, where each index can be an expression.

Examples of indexing a multidimensional array:

Data[2,4] denotes the fourth element in the second array element
Data[2][4] denotes the same element as above.

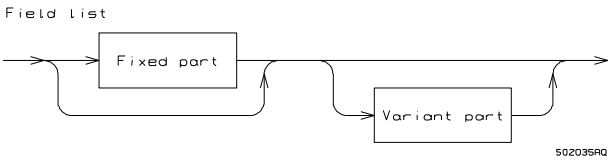
Multidimensional arrays are stored with the right-most dimension increasing first. In the above example this means that the values are stored in the following order: [1,1], [1,2], [1,3], [1,4], [1,5], [2,1], [2,2], [2,3], [2,4], [2,5], [3,1], [3,2], [3,3] and so on.

1,1	1,2	1,3	1,4	1,5
2,1	2,2	2,3	2,4	2,5
3,1	3,2	3,3	3,4	3,5
4,1	4,2	4,3	4,4	4,5

13 Record

A record is a structured data type and as the array type it comprises a set of components. A component in a record is called a field. A field can hold values of a certain type and, unlike array types, each field can be of a different type. The type for a field can be a simple type or a structured type, i.e. an array type or a record type.

The record type declaration specifies the type of each field and the identifier that names the field. The declaration for a record begins with the symbol RECORD and terminates with the symbol END. A field list may contain a fixed part and a variant part.



The fixed part of a record type sets out the list of fixed fields, giving an identifier and a type for each. Each field contains information that is always retrieved in the same way.

Example of a record type:

```
Square = RECORD
           x , y  : INTEGER;
           Area   : REAL;
         END;
```

13.1 Variant part

The variant part of a record type declaration distributes memory space for more than one list of fields, so the information can be accessed in more ways than one. Each list of fields is a variant. The variants overlay the same space in memory, and all fields of all variants can be accessed at all times.

Each variant is identified by at least one constant. All constants must be distinct and of an ordinal type compatible with the tag-field type. Variant and fixed fields are accessed the same way.

Example of a record type with a variant part:

```
Symbol = RECORD
           x, y : INTEGER;
           CASE Figure OF
           Rectangle: (height, width : INTEGER);
           Triangle : (side1, side2, angle : REAL);
           Circle   : (radius : INTEGER);
         END;
```

The record is shown below with the different values for the tag-field.

x	y	Figure	Height	Width	
		rectangle			

x	y	Figure	Side1	Side2	Angle
		triangle			

x	y	Figure	Radius	
		circle	circle	

13.2 Accessing fields

To access a field in a record, the variable identifier for the record type is given first, followed by the field identifier. A point separates the field identifier and the record identifier.

Example of accessing a field in a record type:

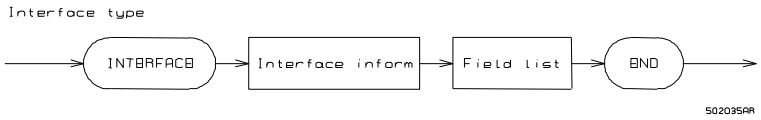
Let FORM be a record of the previous declared type SYMBOL. The fields are accessed in the following way:

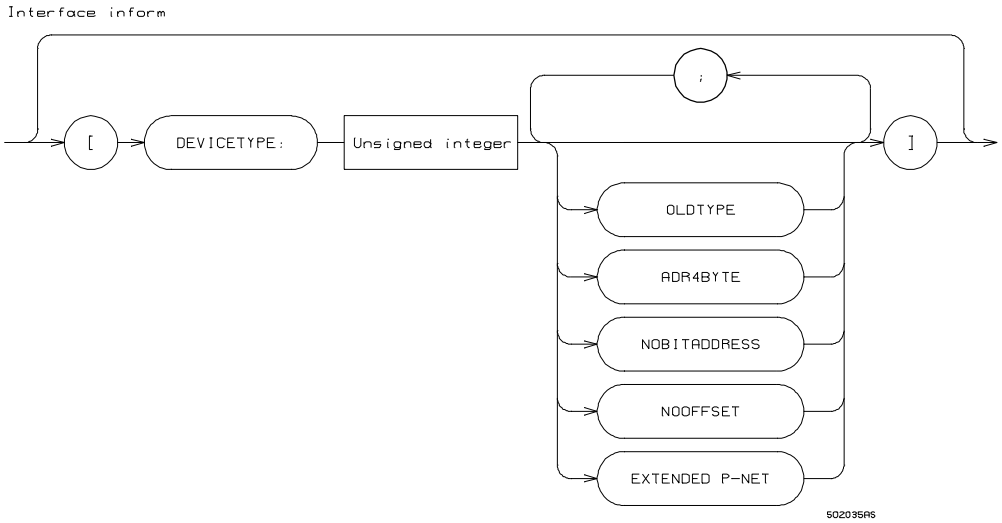
```
VAR
   Form : Symbol;

BEGIN
   Form.x := 25;
   IF Form.x = Form.y THEN ProcesSquare;
   Form.Height := 34;
   Form.Side2 := 12.22;
   Form.Radius := 200;
```

14 Interface

An interface type is used to define an interface module or a channel in an interface module as a whole structured variable. An interface module is constructed with a number of channels, where each channel has 16 accessible registers. The channels can be of the same type or of different types, depending on the specific interface module.

An interface type has a fixed number of components, that can be of different types. An interface type defines a channel, if all the components in the type declaration is of simple type. An interface type defines an interface module, if all the components in the type declaration are of interface type or the type 'Unused'. The first component in the definition of a channel, defines register 0, the second component defines register 1 and so on. The first component in the definition of an interface module, defines channel 0, the second component defines channel 1 and so on.





The interface inform DEVICETYPE is followed by a constant that denotes the module type. DEVICETYPE must be declared.

The interface inform OLDTYPE denotes that the module is of an old type, which means that the variables of real type are stored in a different format. Conversion to the IEEE format is done by the operating system in the controller during program execution and the user will not need to take any action for it.

The interface inform ADR4BYTE denotes the length of the SoftWire No. / abs. address when accessing the module. The length of the address can be 4 byte or 2 byte, denoted by Adr4Byte or Adr2Byte, where Adr2Byte is default.

The interface inform NOBITADDRESS denotes that the module is not able to understand bit addressing.

The interface inform NOOFFSET denotes that the module is accessed with an address without any offset.

The interface inform EXTENDEDPNET denotes that the module understands complex/extended P-NET address, e.g. a controller.

The interface inform NOOFFSETINLONG denotes that the module will not make use of the offset value in a longload or longstore command, i.e. the module calculates the offset value by itself.

Example of an interface type:

```
PD3221 = INTERFACE [ DeviceType: 3221; ObjectType = 1000;
                     Capabilities = NoBitAddress, NoOffsetInLong ]
    Service       : ServiceCh;
    Digital_IO_1        : DigitalCh;
    Digital_IO_2        : DigitalCh;
    Digital_IO_3        : DigitalCh;
    Digital_IO_4        : DigitalCh;
    Digital_IO_5        : DigitalCh;
    Digital_IO_6        : DigitalCh;
    CommonIO    : CommonIO8Ch;
    Analog_In_1 : AnalogInCh;
    Analog_In_2 : AnalogInCh;
    Current_Out : CurrentOutCh;
    PID         : PIDCh;
    Calculator  : CalculatorCh;
    PulseProcessor: PulseProcCh;
   END;
```

14.1 Accessing fields

To access a field in an interface type, the variable identifier for the interface type is given first, followed by the field identifier. The field identifier and the interface variable identifier are separated by a point.

It should be noted that for variables of interface type is it only possible to access one register at a time, and not an entire channel or module.

Example of accessing a field in a variable of interface type:

```
    VAR
    TempModule : PD3221 AT NET: (1,64);

    BEGIN
    IF TempModule.Analog_In_1.AnalogIn >= 45.0 THEN
       OverHeat :=TRUE;
    DigModule.Ch21.Flagreg[7]:=OFF;
    While TempModule.Analog_In_1.AnalogIn >= 35.0 DO
       ChangeTask;
```

See also the examples in the Variable Declaration chapter for how to access variables in external devices.

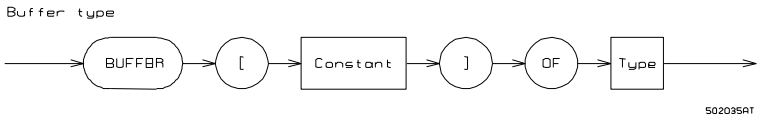
15 Buffer

A buffer is considered as a FIFO (First In First Out), where elements are placed in the end of a queue when a variable is assigned to the buffer, and the elements are removed from the front of the queue, when the buffer is assigned to a variable.

When operating with buffers, the inserting and removing of elements, considers the entire element. This means that if the element type is a structured type, you can not access a specific field directly in the buffer. Instead you must assign the whole element to a variable of the same type and then access the particular field in that variable.

Buffers have a fixed number of elements of one type, the element type. The element type can be of any type except a BUFFER type and a TIMER type.

The syntax for a buffer type is:



The constant denotes the buffer size, the max. number of elements in the buffer.

When an element is read out from a buffer, it is deleted from the buffer and cannot be read out again.

Buffers must always be initiated before they are used the first time. This is done with the standard procedure InitBuffer(buffername).

Before a variable is assigned to a buffer, the program must check if the buffer is full. This is done with BufferFull(buffername) which is a standard function. The function returns a boolean, TRUE if the buffer is full. If a variable is assigned to a buffer and the buffer is already full, an error is produced and the value will not be stored into the buffer, until an element has been removed from the buffer.

Before a buffer is assigned to a variable, the program must check if the buffer is empty. This is done with BufferEmpty(buffername) which is also a standard function. The function returns a boolean, TRUE if the buffer is empty. If an empty buffer is assigned to a variable, an error is produced and the variable will not be assigned until an element has been inserted in the buffer.

If a variable of the type BUFFER is a component of a complex variable, the buffer component variable is only to be used internally in the controller.
(P-NET restriction).

Examples of buffer types:

```
TestVarDef = RECORD
                Var1 :INTEGER;
                Var2 :REAL;
                Var3 :STRING[7];
             END;
   TestVarBuf = Buffer[10] OF TestVarDef;
```

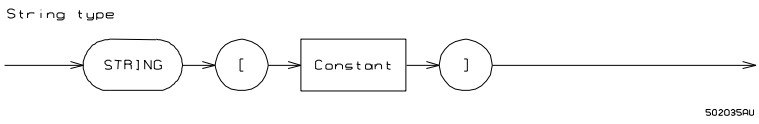
Examples of statements using buffers:

```
InitBuffer(TestVarBuf);

IF NOT BufferFull(TestVarBuf) THEN TestVarBuf:=TestVar;
(* insert an element in the buffer if it is not full *)

IF NOT BufferEmpty(TestVarBuf) THEN TestVar:=TestVarBuf;
(* remove an element from the buffer if there is at least one element *)

WHILE BufferEmpty(KeyboardBuffer) DO ChangeTask;

IF NOT BufferFull(Port_1.OutputBuffer) THEN Port1Output:=HeadLine;

IF NOT BufferEmpty(Port_1.InputBuffer) THEN
   BarCode:= Port_1.InputBuffer;
```

16 String

A string is a sequence of characters with a dynamic length attribute (depending of the actual character count during program execution) and a constant size attribute from 1 to 255.

The syntax of a string type:



A string type can be classified to an array with the following declaration:

```
str = ARRAY[0..MaxStringLength] OF CHAR
```

Characters in a string can be accessed as components of an array.

The length attribute's current value is found in str[0].

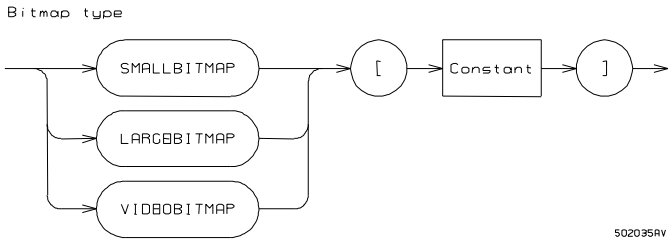
MaxStringLength is a constant in the range 0 to 255.

17 Bitmap

A bitmap defines a pixel-image as a rectangle with a width and a height. The width and height, in pixels, for the symbol is contained as the first elements in a bitmap type and the following elements represents a pixel-image, where each bit is related to a pixel, starting with the most significant bit (bit7). This means that a writing of the bytes in the bitmap, in binary, will present the pixel-image.

A bitmap type is a structured type, characterised by its component type, which is an array of booleans, and a size. A bitmap type is used to create symbols and characters that can be shown on the screen. A charactergenerator is defined as an array of bitmap types.

Process-Pascal has three bitmap types: **smallbitmap, largebitmap** and **videobitmap**.



The size denotes the number of elements (bytes) representing the symbol.

A formula for calculating the size is given by:

```
If width MOD 8 = 0 then a:=0 else a:=1;
size:= ((width DIV 8) + a) * height;
```

17.1 The smallbitmap type

The smallbitmap type defines a bitmap, where the size of the symbol is less than or equal to 255 * 255 pixels (width * height).

The first byte holds the bitmap-width in pixels, and the second byte holds the bitmap-height in pixels.

The smallbitmap has a reference to the pen position on the screen in the upper left corner of the bitmap.

Example of a smallbitmap type followed by a constant declaration:

```
TYPE
        Dottype = SMALLBITMAP[1];

CONST
        Dot = Dottype($01, $01, $80);
```

The example shows a smallbitmap type with a size of '1', pixel-image contained in one byte, and a constant defined as a smallbitmap with width '1' and height '1', which means a single pixel and the pixel is on, i.e. a dot.

17.2 The largebitmap type

The largebitmap type defines a bitmap, with a size for the symbol (width * height), and an offset to a reference point.

The first 2 byte holds the bitmap-width in pixels, and the third and fourth byte holds the bitmap-height in pixels.

The fifth and sixth byte holds an offset to a reference point in the x-direction. The seventh an eighth byte holds an offset to a reference point in the y-direction.

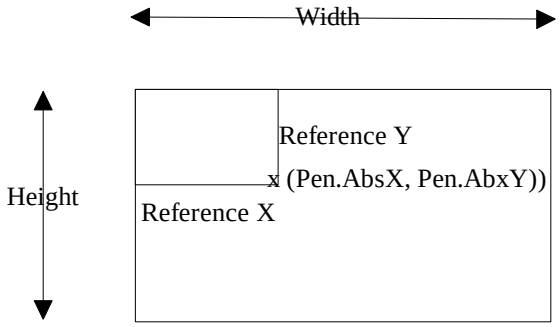
The lowest byte is the MSB for the above mentioned height, width and reference.

The bitmap will be shown with the reference point at the pen position on the screen (Pen.X, Pen.Y).

Example of a largebitmap type followed by a constant declaration:

```
TYPE
        Triangletype = LARGEBITMAP[4];

CONST
        Triangle = Triangletype( $00, $05, $00, $04, $00, $02, $00, $02,
                                 $20, $F8, $70, $20);
```

The diagram shows the reference point (*) corresponding to the pen position for a largebitmap.



17.3 The videobitmap type

The videobitmap type is specially used for defining the video-ram, where the size denotes the capacity for the video-ram. See section SCREEN DEFINITION, how to use videobitmap types.

18 Set

A set type provides a compact structure for recording information about the existence or combination of a collection of values having the same ordinal type.

A set type is a bit array, where each bit indicates whether an element is in the set or not. The maximum number of elements in a set is 256, and a set occupies always 32 bytes of RAM. A set is also a random-access structure whose elements all have the same base type.

A variable of a set type can hold from none to all values of the set.

The base type must not have more than 256 possible values, and the ordinal values of the upper and lower bounds of the base type must be within the range of 0 to 255.

Examples of set types:

```
Smallinteger = SET OF 0..50;
Digit = SET OF '0'..'9';
Letter = SET OF 'A'..'Z';
Colour = SET OF (red, blue, yellow, white, green, black);
```

The order of elements in a set is insignificant and repetition of elements is allowed. The set (3,5..9,2,6) is equal to (2..3,5..9).

Adding new members to a set variable is simply done by adding the ordinal values to the set as follows:

```
ColourSet := ColourSet + [Red, Blue, Green];
```

Removing members from a set variable is simply done by subtracting the ordinal values from the set as follows:

```
ColourSet := ColourSet - [Yellow, Black];
```

The IN operator is used to test for membership of a SET type operand. It returns true when the value of the operand is a member of the set, otherwise it returns false.

Example:

```
IF Blue IN ColourSet THEN Display('Blue is found');

(* test if Blue is a member of the SET variable ColourSet *)
```

19 Userdefined Types

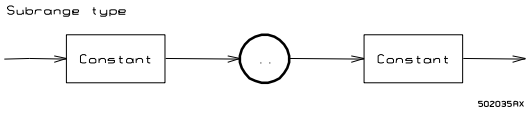
Process-Pascal has a number of pre-declared data types, all described in the previous chapters. From these types its possible to declare new data types.

A user-defined data type is declared in the type declaration part. The name for the user-defined data type is an identifier.

A user-defined data type can contain a previous declared type.

19.1 Subrange types

A subrange type is a range of values from an ordinal type. The definition of a subrange type specifies the least and the largest value in the subrange and includes all values in between these two values.



Both constants must be of the same ordinal type and the first one must be less than or equal to the last one.

A subrange type is mainly used to define an index range in an array structure.

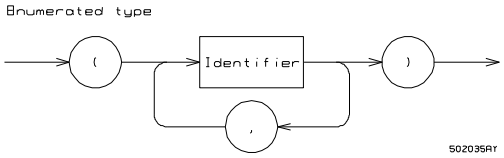
Examples of subrange types:

```
Index20 = 1..20;      (* subrange of Integer *)
Cap_Letter = 'A'..'Z';        (* subrange of Char *)
```

There is no index check on subrange types.

19.2 Enumerated types

Enumerated types define ordered sets of values by enumerating the identifiers that denote these values. Their ordering follows the sequence in which the identifiers are enumerated. Each identifier in the list is declared as a constant for the block in which the enumerated type is declared. This constant's type is the enumerated type being declared.



An enumerated constant's ordinality is determined by its position in the identifier list in which it is declared. The first enumerated constant in a list has an ordinality of 0, the next has ordinality 1, and so on.

Example of an enumerated type:

```
Status = ( Wait, Go, Left, Right, Stop)
```

Given these declarations, **Left** is a constant of type **Status**.

When the **Ord** function is applied to an enumerated type's value, **Ord** returns an integer that shows where the value falls with respect to the other values of that enumerated type. In the example above, Ord(Go) returns 1.

20 Structured Constants

A constant can be a structured type. The declaration of a constant of a structured type specifies the value of each of the elements in the structure.

The structure for a constant can be an array, a record, a set or a string type. Structured constants, which contain the types buffer or timer are not allowed.

20.1 Array constants

The declaration of an array constant specifies, enclosed in parentheses and separated by commas, the values of the components.

Example of an array constant:

```
TYPE
  MonthsType = ARRAY[1..12] OF STRING[3];

CONST
  Months=MonthsType([1]:'Jan',[2]:'Feb',[3]:'Mar',[4]:'Apr',
                    [5]:'May',[6]:'Jun',[7]:'Jul',[8]:'Aug',
                    [9]:'Sep',[10]:'Oct',[11]:'Nov',[12]:'Dec')
```

This example defines an array constant MONTHS, which can be used to print out a 3 character string with the text corresponding to the month number.

If HEADLINE is defined as a string, the following statement

```
HeadLine:=Months[4];
```

will produce the same result as

```
HeadLine:='Apr';
```

Another example of an array constant is a charactergenerator. The standard charactergenerator, named CH6X8.CHR, is an array of bitmaps, where each character is defined as a smallbitmap. The ASCII value for the character is used as an index in the array constant.

```
TYPE
  Character6x8 = SMALLBITMAP[8];
  CG6x8 = ARRAY[$20..$9F] OF Character6x8;

CONST
  Ch6x8 = CG6x8
      ([$20]:($06,$08,$00,$00,$00,$00,$00,$00,$00,$00), (*space*)
       [$21]:($06,$08,$20,$20,$20,$20,$00,$00,$20,$00), (* ! *)
       [$22]:($06,$08,$50,$50,$50,$00,$00,$00,$00,$00), (* " *)
       [$23]:($06,$08,$50,$50,$F8,$50,$F8,$50,$50,$00), (* # *)
       [$24]:($06,$08,$20,$78,$A0,$70,$28,$F0,$20,$00), (* $ *)
       [$25]:($06,$08,$C0,$C8,$10,$20,$40,$98,$18,$00), (* % *)
       [$26]:($06,$08,$60,$90,$A0,$40,$A8,$90,$68,$00), (* & *)
```

and so on.

20.2 Record constants

The declaration of a record constant specifies, enclosed in parentheses and separated by commas, the values of the components.

Examples of a record constant:

```
TYPE
   RecipeType = RECORD
                   SesameSeed   : REAL;
                   RyeFlour     : REAL;
                   Water        : REAL
                END;

CONST
   RecipeDefault = RecipeType(SesameSeed : 10.0,
                              RyeFlour : 65.0,  Water : 25.0);
```

A constant can also be a combination of a record type and an array, as shown in the following example:

```
TYPE
   rec1 = RECORD
          Field1 : INTEGER;
          Field2 : REAL;
   END;

   arr = ARRAY[1..2] OF rec1

CONST
   ArrConst = rec1( [1].Field1: 0, [1].Field2: 2.34,
                    [2].Field1: 4, [2].Field2: 12.40);
```

21 Procedures and Functions

When you are solving a problem and the program increases, it is often convenient to break it into a number of partial problems and solve each partial problem. The concept of PROCEDURES and FUNCTIONS allow you to display each part of your problem in subprograms.

Procedures and functions are useful to use in many situations and as a guide the following should be considered:

1. When a certain sequence of statements are used more than once in the program. This conserves not only your typing time, but also the code size in memory.
2. You should not hesitate from formulating an action as a procedure or a function, even when called only once, if doing so enhances the readability of the program. In general, shorter blocks are easier to understand than long ones.
3. General problems as sorting, print out, weight batching and so on should be solved in a procedure or a function. The CHANGETASK procedure can be called anywhere in a procedure or function, so a single procedure or function can remain active for hours or days without affecting the other tasks.

Procedures and functions can be global or local.

A global procedure or function can be called from more TASKS independently of each other. This means that the same procedure or function can solve a problem for more tasks simultaneously without affecting the other tasks (unless they are using the same global variables).

Before calling a procedure or function within a program, it is required that the procedure or function identifier is declared before it is used. This can in some cases be impossible. To solve this problem, procedures and functions can be "FORWARD" declared. This Forward declaration is an information to the compiler that the identifier needs to be used now, but the declaration will be found later in the program. A FORWARD declaration can be placed anywhere in the program where it is allowed to declare procedures and functions.

Example of a forward declaration:

```
Procedure CloseValves FORWARD;
```

21.1 Procedures

The procedure declaration serves to define a program part and to associate it with an identifier, so that it can be activated by a procedure statement. The declaration has the same form as a program, a heading and a block. Variables declared in a procedure are said to be local variables and these variables are undefined at the beginning of the statement part each time the procedure is activated. Local variables do not exist any more when the procedure is terminated.

Examples of procedures:

```
Procedure CloseValves;
BEGIN
    InletValve[7]:=OFF;
    OutletValve[7]:=OFF;
    ShuntValve[7]:=OFF;
    ValvesClosed:=TRUE
END;

Procedure WaitOneMinute;
VAR
   DelayTimer : TIMER;
BEGIN
   DelayTimer:=60;
   Repeat
         ChangeTask
   Until DelayTimer <= 0;
END;
```

If the procedure must operate on different parameters, these parameters must be introduced in the procedure heading in the procedure declaration. The parameters are listed after the procedure identifier in a formal parameter list.

The parameter list gives the name of each formal parameter followed by its type.

A procedure statement, which states the procedure's identifier, activates a procedure and any parameters required. These parameters are called actual parameters, and are substituted for the corresponding formal parameters that are defined in the procedure declaration. The correspondence between the formal parameters (in the procedure heading) and the actual parameters (in the procedure statement) is established by the positioning of the parameters in the list of actual and formal parameters. Parameters provide a substitution mechanism that allows a process to be repeated with a variation of its arguments.

There are two kinds of parameters: value parameters and variable parameters. The kind of parameters is determined from the formal parameter list in the procedure heading.

21.2 Value parameters

When no symbol heads the parameter section, the parameters of this section are said to be value parameters. In this case the actual parameters (in the procedure statement) must be an expression (of which a variable is a simple case). The corresponding formal parameters represent local variables in the activated procedure. This means that the local variables receive the current values of the actual parameters (the value of the expression at the time of the procedure activation) as initial values. The procedure may then change the value of these variables through assignments, but without affecting the actual parameters. Hence, a value parameter can never represent a result of a computation.

If you are working with large data structures (e.g. an array with a large number of elements), you should be cautious. The copying operation (the value parameters are copied to local variables in the procedure) is relatively expensive in computing time and the amount of data storage needed to hold the copy would be as large as the value parameter (the array). When the procedure terminates, the data storage for the value parameters is released.

21.3 Variable parameters

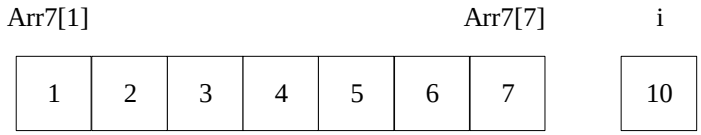
When the symbol VAR heads the parameter section, the parameters of this section are variable parameters. In this case the actual parameters (in the procedure statement) must be variables. The corresponding formal parameters (in the procedure heading) become synonyms for the actual variables during the entire execution. Any operation involving the variable parameters is then performed directly on the actual parameters. The procedure may then change the value of these actual variables through assignments. Hence, a variable parameter can represent a result of a computation.

Example to show the differences of value and variable parameters:

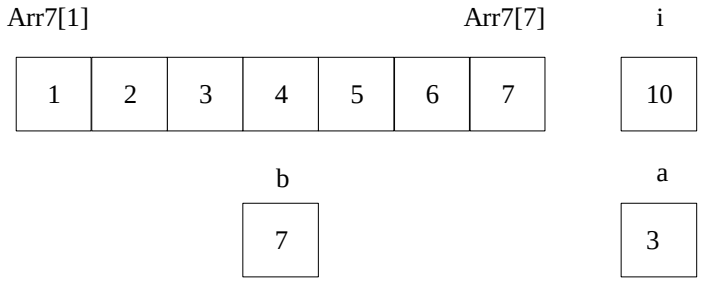
```
TYPE
    Arr7 = ARRAY[1..7] OF INTEGER;
VAR
    Arr : Arr7;
    i : INTEGER;

Procedure ValEx(a,b:integer); (* proc. with value parameters *)
BEGIN
    a:=3;
    b:=7;                                   (* point B *)
END;

Procedure VarEx(VAR a,b:integer); (* proc. with variable parameters*)
BEGIN
    a:=3;
    b:=7;                                   (* point C *)
END;

TASK Example;
BEGIN
    FOR i:=1 TO 7 DO Arr7[i]:=i;
    i:=10;                                  (* point A *)
    ValEx(i,Arr7[4]);
    VarEx(i,Arr7[4]);
END;
```

When the program executes, the data memory will be as follows:
a is initially a copy of i, but the assignment alters it to 3.

Point A:

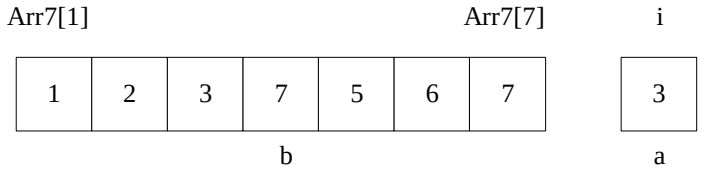


Point B:



b is initially a copy of Arr7[4], but the assignment alters it to 7.

Point C:



Example of a procedure that controls a weight batching with a CHANGETASK procedure statement included:

```
PROCEDURE WeightBatching(    FirstSilo: INTEGER;
                             LastSilo : INTEGER;
                             VAR DataSilo : Silos;
                             VAR DoseValve: IOChannels;
                             VAR Weight: WeightChannel );
VAR
  i : INTEGER;
  DelayTimer: TIMER;
BEGIN
  FOR i:=FirstSilo TO LastSilo DO
  BEGIN
    Weight.Weight1:=0.0;
    DoseValve[i].FlagReg[7]:=On;
    REPEAT
      WeighOut:=Weight.Weight0;
      ChangeTask;
    UNTIL DataSilo[i].WeighOut <= DataSilo[i].Setpoint-DataSilo[i].Tails;
    DoseValve[i].FlagReg[7]:=Off;
    DelayTimer:=5;
    WHILE DelayTimer >= 0 DO
    BEGIN
      ChangeTask;
      DataSilo[i].WeighOut:=Weight.Weight0
    END
  END
END;
```

21.4 Functions

Functions are program parts (in the same sense as procedures) which compute a single ordinal or real value for use in the evaluation of an expression. The declaration has the same form as a program, a heading and a block.

The function heading specifies the identifier for the function, the formal parameters (if any), and the function result type. The result data type for a function can only be a simple type. The variable and value parameters are discussed in the previous section PROCEDURES.

A function call denotes the function's identifier and any actual parameters required by the function. A function call appears as an operand in an expression. When the expression is evaluated, the function is executed, and the value of the operand becomes the value returned by the function.

A function is generally used when you only need to return a single value, the function value.

The block within the function declaration should contain at least one executed assignment statement that assigns a value to the function identifier. This assignment returns the result of the function. The result of the function is the last value assigned before the function terminates.

Example of a function:

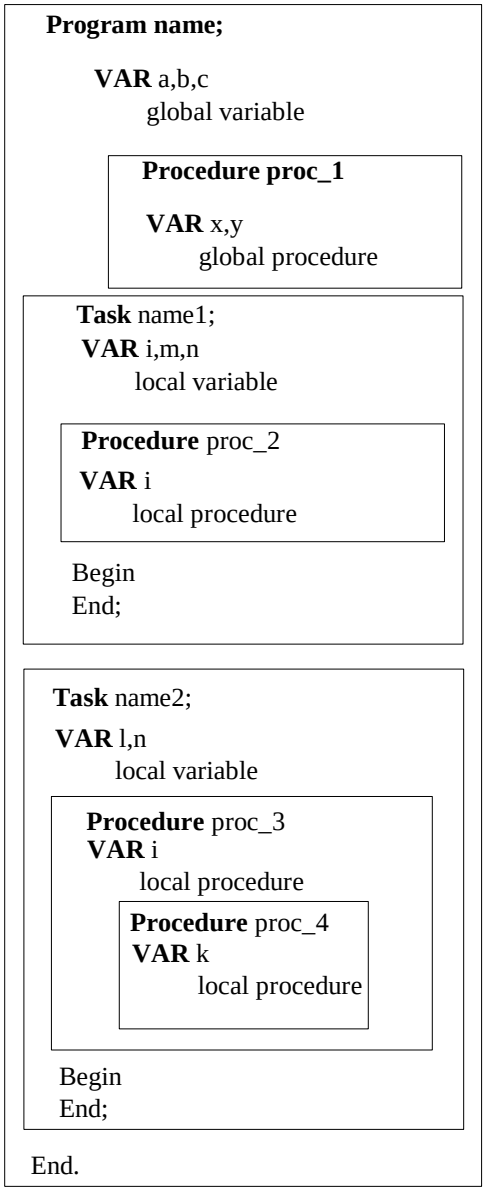
```
FUNCTION Max(VAR a:IntegerArray):INTEGER;
VAR
   i, x : INTEGER;

BEGIN
   x:=a[1];
   FOR i:=2 TO 10 DO
      IF x < a[i] THEN x:=a[i];
   Max:=x
END;
```

The function returns the largest value in an array and it is called in a statement by its identifier.

```
IF Max(Numbers) > 100 THEN .....

LargestNumber:=Max(Numbers);
```

22 Scope.

In Pascal, all identifiers needs to be declared before they can be used or accessed. This means that an identifier is accessible in the block in which is has been declared and in the following blocks.



Only the variables **a, b, c, x and y** are available for **proc_1**.

The variables **a, b, c, i, m and n** are available for task **name1**.

The variables **a, b, c, i, m and n** are available for **proc_2**, where the variable **i** is the local variable for the procedure. The variable **i** for the task is not available for **proc_2**.

The variables **a, b, c, l and n** are available for task **name2**. **n** is not the same as **n** in task **name1**.

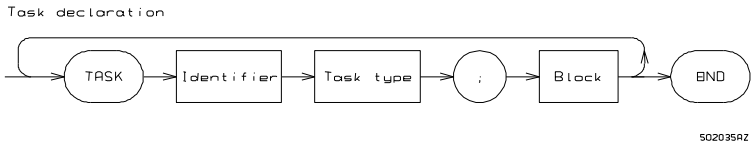
The variables **a, b, c, l, n and i** are available for **proc_3**.

The variables **a, b, c, l, n, i and k** are available for **proc_4**.

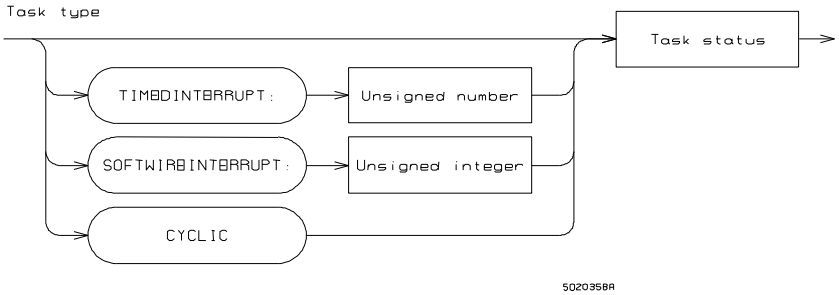
23 Task Declaration

Process-Pascal handles 3 different types of TASKs: CYCLIC TASK, - TIMEDINTERERUPT TASK and SOFTWIREINTERRUPT TASK. All 3 types of TASK can be used within the same program.

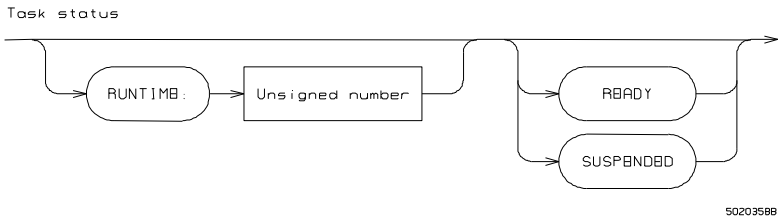
The task declaration serves to define a program part and to associate it with an identifier. The declaration has the same form as a program, a heading and a block.



The task heading names the task's identifier and specifies the task type.



A task declared without a task type attribute is given the default task type CYCLIC.



The max RUNTIME is given by a real constant and is declared in seconds and the resolution is 1/128 second. This can be used to force a changetask in the task to ensure that it will not prevent the other tasks from running if it enters a "loop forever" without a changetask statement in the loop. The default RUNTIME is 300 seconds.

The max RUNTIME can be changed during program execution with the standard procedure MAXRUNTIME(time), where time must be a constant or a variable, denoting the new max runtime in seconds.

Below is an example on a timed interrupt task, that runs every second:

```
TASK ProcessTime TIMEDINTERRUPT: 1.0;
VAR
    Hour, Min, Sec: Integer;
BEGIN
    Hour:=0; Min:=0; Sec:=0;
    REPEAT
        Sec:=Sec+1;
        IF Sec = 60 THEN
        BEGIN
            Sec:=0;
            Min:=Min+1;
            IF Min = 60 THEN
            BEGIN
                Min:=0;
                Hour:=Hour+1;
            END;
        END;
        ChangeTask;
    UNTIL Not Process_On;
END; (* Task ProcessTime *)
```

The initial task status can be declared to be READY or SUSPENDED. The default task status is READY.

All the tasks are grouped in a task chain system. The cyclic tasks are in a cyclic ordered chain with one task pointing to the next cyclic task, where CHANGETASK switches to the next one in the chain. The order of the cyclic tasks in the chain is defined by the order of the TASKs in the program but this order can be changed if any of the tasks change status during the program.

The timed interrupt tasks are in another chain where the order of the tasks is determined by the next time they must run. TimedInterrupt TASKs are executed with certain time intervals, controlled by the programmer. The time period is given by the real constant and is declared in seconds with a resolution of 1/128 second.

The softwire interrupt tasks are in a third chain determined by the corresponding interrupt connections. The interrupt connections and interrupt conditions are declared in the global variable declaration.

When a task is found in a chain, the task status is READY. A SUSPENDED task has been removed from the task chain system and will not be able to run as long it is suspended.

A SUSPENDED task can change to READY status, if another RUNNING task calls the standard procedure CONTINUETASK with the appropriate task identifier, ContinueTask(TaskIdentifier). This will insert the task in the appropriate task chain again and make the task "TaskIdentifier" continue from where it was last stopped or interrupted.

A READY task can change to SUSPENDED status, if another RUNNING task calls the standard procedure STOPTASK with the appropriate task identifier, StopTask(TaskIdentifier).

This will remove the task from the task chain system and prevent the task "TaskIdentifier" from running any more, until it is changed to READY again from another task by means of CONTINUETASK(TaskIdentifier) statement.

If a task comes to an END for the task block, its status is automatically changed to SUSPENDED and the task is removed from the chain and the task program counter is set to the beginning of the task. Consequently the task will RESTART, if another task calls the CONTINUETASK(taskidentifier) statement.

A RUNNING task can force itself to RESTART from the beginning of the task. To perform a restart for a task, the standard procedure RESTARTTASK is called. After calling RESTARTTASK, the program execution will continue with the first statement within the task.

To allow starting and stopping task within a program, it is required that the task identifiers are declared before they are used. This can in some cases be impossible. To solve this problem, a FORWARD declaration of task identifiers can be included. A FORWARD declaration can be placed anywhere in the program where it is allowed to declare task.

Example of a forward declaration:

```
Task WeightControl FORWARD;
```

The time interval for a timed interrupt task can be changed during program execution by means of the standard procedure TIMEDINTERRUPTTIME(time), where time can be a constant or a variable, denoting the interval time in seconds. The time is specific for the task, which calls the procedure so the procedure must be called from the task where the time must be changed, i.e. a task can only change it's own time. When a task is declared to be a timed interrupt task, the time interval must be declared in the task heading.

Example of a task heading for a TimedInterrupt task:

```
TASK ProcessTime TIMEDINTERRUPT: 1.0;
```

For cyclic tasks, TIMEDINTERRUPT TASKs can be ENABLED, i.e. allowed to interrupt, or DISABLED, not allowed to interrupt in cyclic tasks. ENABLE(-TimedInterrupt) is a standard procedure, to be used in **cyclic** tasks, that allow timed tasks to interrupt the cyclic task. In all cyclic tasks, TIMEDINTERRUPT TASKs are ENABLED as default after a reset.

DISABLE(TimedInterrupt) is a standard procedure to be used in cyclic tasks, that disables all TIMEDINTERRUPT TASKs, i.e. denotes that no timed interrupt task are allowed to interrupt this cyclic task. Disabling the timed interrupt tasks will not change the status of these tasks. This means that they are not removed from the task chain, and when the timed interrupt tasks are enabled again, the timed interrupt tasks will try to catch up with the lost time (if any). If timed interrupt is disabled or enabled in a procedure or in a function, the interrupt status is automatically set back to the state it was before the call when the procedure or function is exited.

SOFTWIREINTERRUPT TASKs are executed each time a certain global defined variable is accessed. The conditions for activating the interrupt when accessing the variable are set by the variable declaration.

See the chapter INTERRUPT for connecting an interrupt to a variable. A number in the range 0 to 31 gives the interrupt connection. Several global variables may be connected to the same interrupt number.

Example of a task heading for a softwireinterrupt task:

```
TASK Keyboard SOFTWIREINTERRUPT:0;
```

The different tasks can **change task type** during program execution.

A CYCLIC task and a SOFTWIREINTERRUPT task can change to a TIMEDINTERRUPT task by means of the standard procedure TIMEDTASK. Before changing the task type to TimedInterrupt, the interval time must be selected, TimedInterruptTime(time), or a default value of 255 seconds is used. Changing the task type, will not affect the program execution and the task will continue until it meets a ChangeTask statement.

A TIMEDINTERRUPT task and a SOFTWIREINTERRUPT task can change to a CYCLIC task by means of the standard procedure CYCLICTASK. Changing the task type, will insert the task in the cyclic sequence and the program execution for the task will continue until it meets a ChangeTask statement. When the task runs again, it is within the cyclic sequence and continues in the statement after ChangeTask.

A CYCLIC task and a TIMEDINTERRUPT task can change to a SOFTWIREINTERRUPT task, by means of the standard procedure INTERRUPTTASK, but only if the task originally was declared as a softwireinterrupt task. The interrupt connection is set to the initial softwireinterrupt number (declared in the task head). Changing the task type, will not affect the program execution and the task will continue until it meets a ChangeTask statement. The task will run again in the next statement when an interrupt with the corresponding interrupt connection is generated.

24 Interrupt

Process-Pascal offers you facilities for generating interrupts and executing interrupt tasks. You can have 32 different interrupts, each denoted by a number in the range from 0 to 31.

An interrupt is generated on accessing a specific global variable, which has been declared with a softwireinterrupt connection, given by an interrupt number. Interrupt can only be generated on internal variables. A task can be declared as a softwireinterrupt task with an interrupt connection. This means that an interrupt task with an interrupt number is executed when the variable with the same interrupt number is accessed.

The interrupt condition for accessing the variable is set to “any access" as default. The interrupt condition could be specified to be one or several of the following: INTERNLOAD (the controller itself loads the variable), INTERNSTORE (the controller itself stores a value in the variable), EXTERNLOAD (the variable is loaded via the P-NET from another controller or PC) or EXTERNSTORE (a value is stored in the variable via the P-NET).

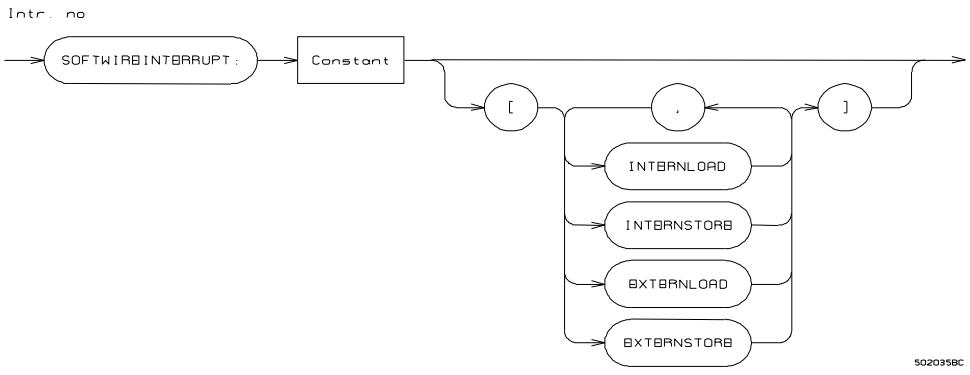
If more interrupts occur at the same time, the corresponding interrupt tasks will be executed in priority according to the interrupt number, i.e. the highest number will have the highest priority.

Example of a variable declaration with interrupt:

```
VAR
        KeyboardBuffer :Buffer[10] OF BYTE SOFTWIREINTERRUPT:0
                                    [INTERNSTORE, EXTERNSTORE];
```

The above declaration of the keyboardbuffer connects the variable to interrupt number 0. The interrupt condition is set to any internal or external store in the variable. The priority of this interrupt is set to the lowest priority.

Softwire interrupts can be ENABLED, i.e. allowed to interrupt, or DISABLED, not allowed to interrupt in cyclic tasks. ENABLE(SoftwireInterrupt) is a standard procedure to be used in cyclic tasks, that allow softwire interrupt tasks to interrupt the cyclic task. In all cyclic tasks, SOFTWIREINTERRUPT TASKs are ENABLED as default after a reset. If interrupt is disabled or enabled in a procedure or in a function, the interrupt status is automatically set back to the state it was before the call when the procedure or function is exited.



DISABLE(SoftwireInterrupt) is a standard procedure that closes for any softwire interrupt in a task. The variables with an interrupt connection are not affected by DISABLE, but if a variable has been accessed with an interrupt condition while the interrupt is disabled, the corresponding interrupt task will be activated if interrupt is enabled again.

It is possible to relate a variable of type BUFFER to another variable with an interrupt connection. Each time an interrupt related to the variable is generated, an element is stored in the buffer variable. The buffer element holds information on the SWNo which caused the interrupt (there might be more variables with the same interrupt) and an offset in bytes to the part of the variable which was accessed.

The buffer element is of the following type:

```
IntRecordType =     RECORD
                        SWNo  :  INTEGER;
                        Offset   : INTEGER;
                    END;
```

Example for connecting the buffer to an interrupt variable:

```
VAR
    IntBuffer:          Buffer[10] OF IntRecordType;
    DataBase :          DataBaseType SOFTWIREINTERRUPT: 3
                            [ExternStore, InternStore] IntBuffer;
```

25 WHEN ERROR

Some build in facilities in Process-Pascal gives you the possibility for using data distributed on the P-NET fieldbus system. To secure your program against any erroneous data, as well internal as external in the system, Process-Pascal offers an automatic error detecting system.

When using a network to communicate with interface modules or other controllers, here P-NET, errors can occur. The possible errors that can appear are called INTERFACE ERRORS and can be transmission errors relating to the network or data errors relating to the interface modules. An error in a module can be related to the whole module or a single channel.

Executing a program can generate some different run-time errors, caused by the operator or the programmer. These errors will NOT stop program execution but generates an error code.

The automatic error detecting system is enabled by a WHEN ERROR THEN statement. This statement should be followed by a section of statements, which handles the error condition, i.e. closing valves or stopping production. This section of program will only be executed if an error occurs in the succeeding part of the task.

The WHEN ERROR THEN statement is task dependent, meaning that the automatic error detecting system is only enabled for the tasks that have executed a WHEN ERROR THEN statement.

The figure below illustrates the structure for a task using WHEN ERROR:

```
Task name
VAR
   local variable

Procedure
   local procedure

Begin
   statements for this task    (* ref.1 *)

   WHEN ERROR THEN
   Begin
      statements for handling errors
   End;
   statements for this task     (* ref.2 *)
End;
```

Task Heading.

Local variable declaration.

Local procedure declaration.

Task statements.

Error handling part.

Task statements.

If an error occurs in the first part of the task (* ref.1 *), this will not affect the program execution, but erroneous data can be loaded and may cause problems, i.e. in calculations.

The error handling part of the program is defined in the section after the WHEN ERROR THEN statement. If an error occurs in the last part of the task (* ref.2 *), this will interrupt the program execution in the statement which caused the error, and move the program execution to the error handling part after WHEN ERROR.

The error handling part after WHEN ERROR THEN can end in three ways:

1. the program continues with the statements after the error handling part,
2. the program execution can RETURN to the statement where the error occurred and continue from there, i.e. proceed with the next P-code,
3. the program execution can return to the statement where the error occurred and retry the P-code.

To make the program execution return, a standard procedure *RETURN* must be called.

WARNING: When using RETURN, the program execution continues in the P-code AFTER the one in which the error occurred and there is a risk of erroneous data in the succeeding calculations.

To make the program retry the P-code that caused the error, a standard procedure *RetryIfLegal* must be called.

WARNING: When using RetryIfLegal, the program execution retries the P-code in which the error occurred and there is a risk of an infinite loop or a very slow system in case of many errors. If using th RetryIfLegal procedure, you should always implement a counter and a maximum value for the counter to avoid that your program locks. The RetryIfLegal procedure can only be executed if the "WHEN ERROR program" was invoked by a transmission error.

To enable, disable, clear and test various error states, corresponding to a number of error bits, some standard procedures/functions are available in Process-Pascal:

25.1 WHEN ERROR THEN [Disable]

The WHEN ERROR THEN statement activates the automatic error detecting system and enables for all error conditions, i.e. enables all error bits. When an error occurs, program execution is interrupted and moved to the WHEN ERROR part. The [Disable] parameter is optional and makes it possible to disable for changetask (interrupt from other tasks) to protect the program execution in the WHEN ERROR section. If [Disable] for ChangeTask is used in the "WHEN ERROR program", ChangeTask is automatically enabled on exit of the WHEN ERROR program, either by "Return", "RetryIfLegal" or "End".

A bit specification can be used to specify the error bits to be cleared, disabled, enabled, raised or tested.

The different errors to clear, disable, enable, raise and test are:

PnetError,	HisError,ModuleError,	ActError,DataError,	
BufferError,	ArithmicError,	IndexError,	ConvertError

The first three errors are caused by external events:

PnetError	corresponds to transmission error on the P-NET,
HisError,ModuleError	corresponds to a historical error or a module error in the accessed module, i.e. the ChError.His register is not 0,
ActError,DataError	corresponds to an actual error on the data, a data error in the accessed module, i.e. the ChError.Act register is not 0.

The next four errors are caused by internal events:

BufferError	a buffer is accessed when it is full/empty,
ArithmicError	division by zero, over-/underflow,
IndexError	array index out of bounds,
ConvertError	error in converting ASCII to numeric.

These last four errors also generate an error code in the controller errorcode.

BITTEST

Bittest is a function used for testing error bits, generated by the automatic error detection system or the P-NET operating system. The function returns a boolean.

```
BitTest(Error [,errorbit, .., errorbit]);
```

Using Bittest on ERROR allows you to test error bits, generated by the automatic error detection system. If the bit specification is omitted, bittest is true if one of the errors are true, otherwise the specified errors are tested.

NOTE: to ensure testing of the right error bits, the error bits must be cleared after the WHEN ERROR part, since the operating system will not clear these bits.

```
BitTest(Transmission, TransmissionErrorBit);
```

Using Bittest on TRANSMISSION allows you to test error bits, generated by the PNET operating system. Bittest is true if the corresponding error bit is true. The error bit corresponds to the bits in the fieldvariable ErrorCode from the InterFaceErrorBuffer (see the following pages).

CLEAR

Clear is used to clear error bits, generated by the automatic error detection system. If the bit specification is omitted, all error bits are cleared, otherwise the specified error bits are cleared.

```
Clear(Error [, errorbit, .., errorbit])
```

DISABLE

Disable is used to disable all errors or specific errors, generated by the automatic error detection system. Disabling error bits will prevent the WHEN ERROR part to be executed when the corresponding errors occur.

```
Disable(Error [, errorbit, .., errorbit])
```

ENABLE

Enable is used to Enable all errors or specific errors, to be generated by the automatic error detection system.

```
Enable(Error [, errorbit, .., errorbit])
```

RAISE

Raise is used to force an error state, ignoring the automatic error detection system. An error can be raised in a specific task denoted by TaskIdentifier, or the error is raised within the task, which called the Raise procedure.

```
Raise([TaskIdentifier, ] Error [, errorbit, .., errorbit])
```

25.2 ERROR REPORT

When an error is detected by the operating system, the operating system can assign a number of parameters, collected in a report, to the global variable called Inter-FaceErrorBuffer. This variable is declared in the system file for the controller in question as a buffer with 10 elements. Each element is defined as a record of 4 fields, containing information of the variable, which caused the interface error. Three different errors can cause the operating system to produce this report, denoted by the following identifiers:

```
PnetReport,        HisReport,        ActReport.
```

These report bits can be disabled or enabled independently by means of Disable(Error,reportbit) or Enable(Error,reportbit). The WHEN ERROR statement enables all three report bits and all error bits.

PnetReport	only communication errors on the P-NET will insert an element in the InterFaceErrorBuffer,
HisReport	only historical errors in the accessed module will insert an element in the InterFaceErrorBuffer. *ModuleReport* can be used instead of *HisReport*
ActReport	only data errors in the accessed module will insert an element in the InterFaceErrorBuffer. *DataReport* can be used instead of *ActReport*.

The declaration of the InterFaceErrorBuffer is shown below.

```
TYPE
   InterFaceErrorRecord =  RECORD
                             SWNo     : WORD;
                             VARAddr  : LONGINTEGER;
                             VAROffset: WORD;
                             ErrorCode: WORD;
                                   END;

VAR
   InterFaceErrorBuffer   : BUFFER[10] OF InterFaceErrorRecord;
```

Since the variable InterFaceErrorBuffer is of type buffer, it is not possible to read a field in an element. You must declare a new variable of the same type as the elements in the buffer. When doing so, you can read the entire element and then access each field in the new variable.

See also the example in the procedure WhenErrorRoutine or the task Error_In_InterFace how to use InterFaceErrorBuffer.

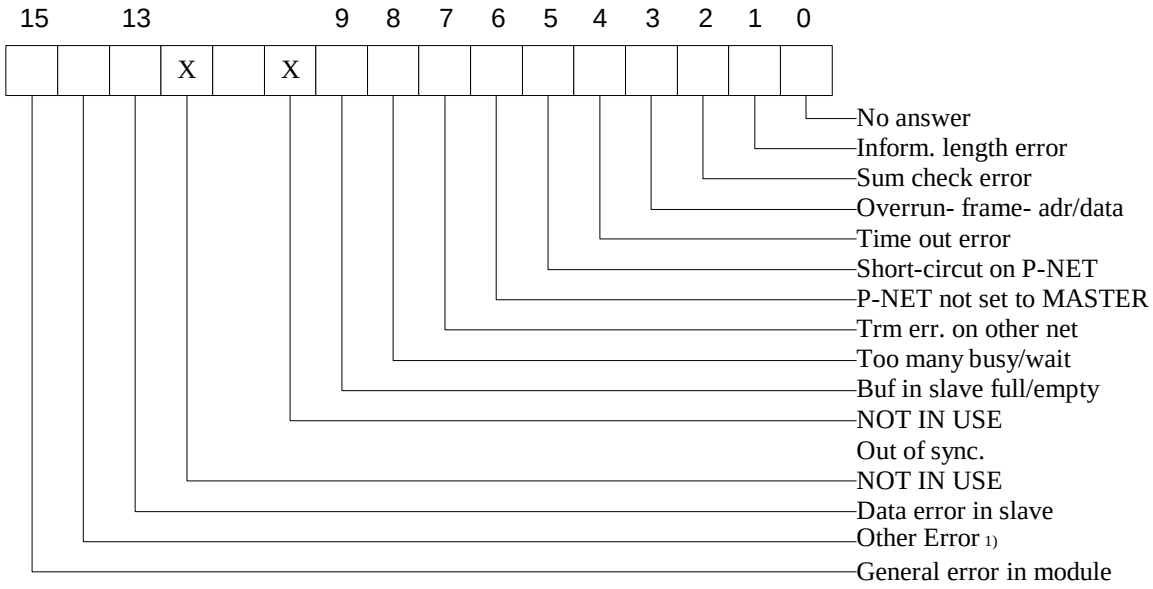
NOTE: When activating the automatic error detecting system and a report element is stored in the buffer, relevant program must be written to read this report element from the InterFaceErrorBuffer to prevent the buffer in running full.

The fieldvariable **SWNo** holds a SOFTWIRE number for the interfacemodule variable, which caused the interfaceerror. The standard function **VARNAME(SOFTWIRENo)** returns the stringconstant after NAME for the module variable, if it is declared. See also the chapter VARIABLE DECLARATION how to assign a name string to a variable.

The fieldvariable **VARAddr** holds a logic address in the interfacemodule for the variable. For simple interfacemodules (I/O modules), the contents of VARAddr is a number which combines the channel number and the register number for the variable. If the module is a controller, VARAddr holds the SOFTWIRE number for the variable in the other controller, which caused the interfaceerror.

The fieldvariable **VAROFFSET** holds an offset for the variable (in the interfacemodule) which caused the interfaceerror. The field variable VAROffset can be used to locate a variable in a complex variable.

The fieldvariable **ErrorCode** holds the errorcode related to the interfaceerror. The field is declared as a word, where each bit has the following meaning:



1) Data format error, SWNo error, Write protection, Node addr. error, Instruction error

If ErrorCode is greater than $8000, then (ErrorCode AND $7FFF) denotes the index in the NodeList array that were used with PointerToNode to access the variable.

NOTE: The error codes in a PD 5000 Controller are interpreted in a different way. Please refer to the PD 5000 Manual (ref. No. 502 077) for the exact details. The PD 5000 error codes can be converted to the above format by calling the standard function *ConvertErrorCode*.

A characteristic structure of the error handling section is show in the example below:

WHEN ERROR THEN [Disable]
BEGIN (* The error detection is automatically disabled when the WHEN ERROR part is entered. This prevents the error program to enter a loop forever if new errors occurs during the error handling procedure. *)

(* Procede your error handling procedure , ie. closing valves or stop production. The error handling procedure should bring your process back to a well defined state, from which it can continue. You may use RETURN or RETRYIFLEGAL to return to the program section where the error was detected *)

END; (* End of the error handling procedure. The error detection is automatically enabled again and will call this error handling section when the next error occurs. The program will continue with the statement followed by this "END" *)

Below is an example of the WHEN ERROR statement using RETURN:

```
WHEN ERROR THEN [Disable]  (* disable Changetask *)
BEGIN
    WhenErrorRoutine(GlobalErrorString);
    Enable (ChangeTask);
    Return;
END;
Disable(Error);
Enable(Error, PnetError, HisError, PnetReport, HisReport);
```

The example above will call e common error handling routine, that returns the error information in a global error string. Please refer to the files “When_Error.inc” in the Process-Pascal library for additional details.

It is also possible to connect an interrupt to the InterFaceErrroBuffer and the call a common task, which is activated on an interfaceerror by a softwireinterrupt. The interrupt is connected to the variable INTERFACEERRORBUFFER as mentioned on the previous pages. Please refer to the file “Intererr.inc” in the Process-Pascal library for additional details. The task generates a string, containing an error message. The ErrorText variable is assumed to be a global string.

The examples in the Process-Pascal library are using a common procedure to find the error information.

25.3 ERRORCODES

Some of the above mentioned errors can set an error code in the controller. Please refer to the manual for the controller in question to get a complete list of errors and the relating error codes.

26 The SoftWire List

The SoftWire list is synonymous with a list containing information of the "wiring" of the plant. It has a table-like construction. The compiler converts the global identifiers used in Process-Pascal into a number. These SoftWire numbers are used as an entry key to the softwire list which contains structured information on each individual global variable and constant, used in the particular program.

The SoftWire list contains information of the following:

1. P-NET number and type of the unit - possibly internal - where the variable is stored.

2. The data type of the variable, such as integer, real, array, record, etc. It is worth mentioning that a record can represent a complete channel in a P-NET module.

3. The address of the variable. If the variable is available internally, the list contains a physical address, whereas the list contains a softwire number or logic address if the variable is external.

4. Name of alarm. In case an error occurs concerning a variable, for instance an analog measuring channel, the name of the alarm will be included in the automatic error report (application program, written in Process-Pascal).

The SoftWire list is generated by the Process-Pascal compiler, based on the global variables declared in the Process-Pascal program. The contents of the SoftWire List can be seen in the MAP file.

During compilation, the compiler also generates a list of SoftWire numbers, which denotes all the devices and channels that are declared within the program. This list is stored in a global constant called PDBoxDefinition.

During starting up of the program or during configuration of the plant, it is possible to check that all the connected units are available and are equivalent to the types specified in the softwire list (optional program to include in the application program).

26.1 The aim of the SoftWire list.

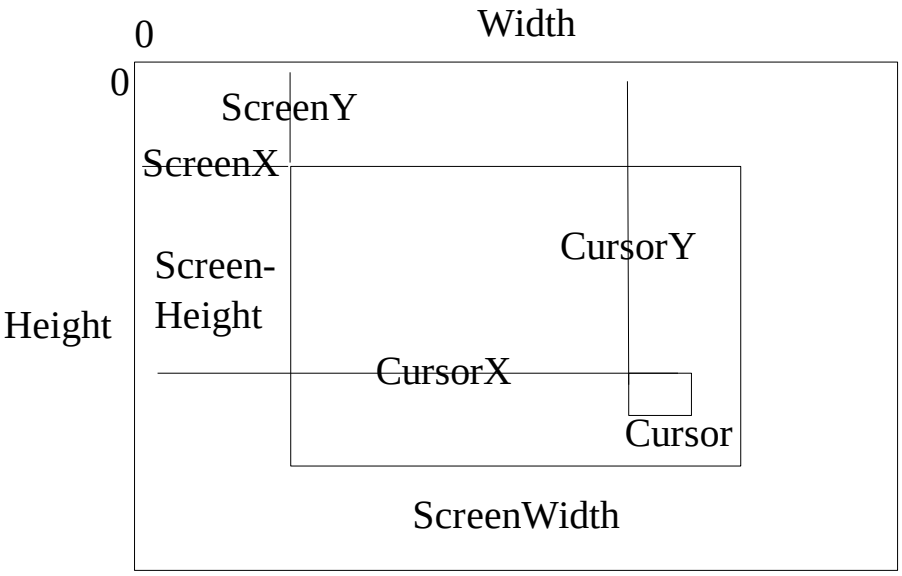
The SoftWire list makes the global variables, declared in the individual controllers, applicable all over the network, thus enabling several controllers to co-operate.

If data are to be sent from one controller to another, no identifiers are known in the other controller and consequently data and identifiers must be related to some numbers in a list, the SoftWire List.

The number of the P-NET interface modules and the addresses of the variables can be changed within the SoftWire list without re-compiling the programs of the other controllers.

27 Screen Setup and Definition.

A picture can be defined to a certain size (ScreenInfo.Width * ScreenInfo.Height) in pixels. The screen shows a section of the picture, defined by the upper left corner by ScreenInfo.ScreenX and ScreenInfo.ScreenY. This section, the basic window, is set by the standard procedure SetWindow.



The system files for the various controller types declare a variable called ScreenInfo, which is a record that holds information of the picture and the screen and is of the following type:

```
ScreenInformationType = RECORD
        Video: BitMapPtr;               screen definition
        Width: INTEGER;
        Height: INTEGER;
        CursorX: INTEGER;               cursor definition
        CursorY: INTEGER;
        CursorForeGround: BYTE;
        CursorBackGround: BYTE;
        Cursor:  BitMapPtr;
        ScreenX: INTEGER;                       basic window definition
        ScreenY: INTEGER;
        ScreenWidth: INTEGER;
        ScreenHeight: INTEGER;
    END;
```

Video holds a pointer the screen (the video RAM). It is not possible to access the display directly via this pointer. The pointer is set up by the standard procedure *SetScreen*.

Width and *Height* defines the width and height of the screen, in pixels. The values are set up by the standard procedure *SetVideo(x,y)*.

CursorX, *CursorY* defines the actual position of the cursor. It is used for reading only (the cursor cannot be moved by writing to these values). The cursor position is changed by means of the standard procedures *CursorToAbs(x,y)*, *MoveCursor(x,y)* or *CursorTo(x,y)*. Refer to Process-Pascal manual for further details.

CursorForeground, *CursorBackground* defines the foreground and background colors of the cursor. The colours can be accessed directly, or set by the standard procedure *SetCursorColors*. If the colours are accessed directly, the cursor will not change colours until it is moved.

Cursor holds a pointer to the cursor bitmap. The value of this variable is for internal use only. The pointer is set up by means of the standard procedure *SetCursor*.

ScreenX and *ScreenY* are used in controllers with more windows. The standard procedure *SetWindow(x,y)* will insert x in *ScreenX* and y in *ScreenY*. Refer to Process-Pascal manual for further details.

ScreenWidth, *ScreenHeight* holds the physical width and height of the screen in pixels.

To access the picture, a corresponding videoram is declared as a variable informing about the size and address for this videoram as shown below:

```
VAR
  Picture : VIDEOBITMAP['picturesize'] AT ADDRESS: $'adr';
```

where 'picturesize' is size for the actual picture in bytes.
'adr' is the address for the videoram for the actual picture

Example:

```
                    Screen : VIDEOBITMAP[$4000] AT ADDRESS: $4C0001;
```

The standard procedure **SETSCREEN** selects a variable of the type **VIDEOBITMAP** for the actual picture and a pointer is generated to the field variable ScreenInfo.Video.

A variable of type BitMapPtr is used by the operating system to locate the variable in memory. All variables of type BitMapPtr should not be accessed in the program by the user because they are changed and used by the standard procedures with the name SET....., i.e.: SETVIDEO.

The standard procedure **SETVIDEO** clears the picture to background color and passes its parameters to ScreenInfo.Width and ScreenInfo.Height and sets these parameters to the videocontroller. Assigning values to ScreenInfo.Width and ScreenInfo.Height has no influence on the picture, unless **SETVIDEO** is called.

The cursorposition is a pixel position calculated relatively from the picture start and is given by ScreenInfo.CursorX and ScreenInfo.CursorY. Picture start is the pixel position (0,0) (upper left corner).

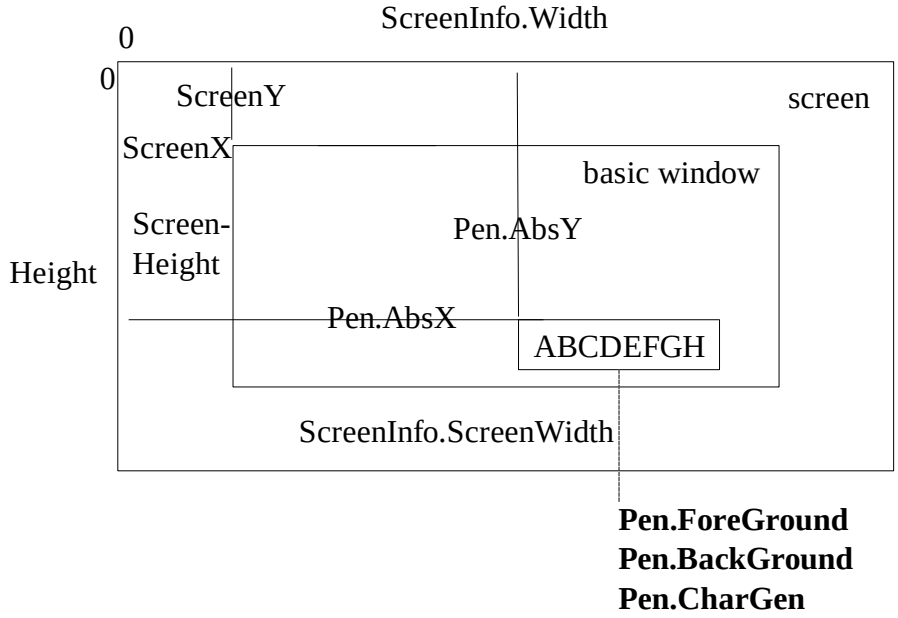
The standard procedure **SETCURSOR** selects a variable of the type bitmap for the actual cursor and a pointer is generated to the variable ScreenInfo.Cursor. The pointer for ScreenInfo.Cursor must be generated at least once in the program, if a cursor is used within that program. The cursor is NOT used for writing at the screen, it is only used to point out variables at the screen when the user want to change their value from the keyboard.

The standard procedure **SETWINDOW** selects a section of the picture to be shown on the screen, the basic window. The window is defined by ScreenInfo.ScreenX and ScreenInfo.ScreenY as the upper left corner of the screen, where ScreenInfo.ScreenWidth and ScreenInfo.ScreenHeight must be set to the actual number of pixels in the used hardware.

28 Writing on the Screen.

Writing on the screen can be done with different characters and symbols in various sizes, all independent of each other and at the same screen. This means that it is simple to combine text and graphics because all text is written in graphic mode. All texts and symbols can be placed on the screen in any pixel position, so text can be written in proportional writing and with any line spacing.

All what is written on the screen, is written in windows. First the basic window is selected (the window is automatically opened by the selection) and following a number of windows can be opened from that basic window (only used in PD5020). The following description is only concerning writing in the basic window.



When you want to write at the screen, two standard procedures are available: **Display** and **Update.** When these procedures are called, different parameters must be passed to the procedures. One of these parameters, the first one, holds information of the charactergenerator, writing-mode and penposition.

Writing on the display always requires a pen. If no pen is mentioned in the statement for writing - e.g. *Display()* - *DefaultPen* is used as default. If a local *DefaultPen* is declared, it will be used - otherwise the globally declared *DefaultPen* will be used. The pen holds information on charactergenerator, colours, window number, pen position etc.

Before the above mentioned procedures are called, a charactergenerator, a foreground- and a background-colour and a pen-position must be assigned to the variable of type PenInformationType.

A Pen is declared as a record of the following type:

```
Record
    CharGen       : CharacterGeneratorPtr;
    ForeGround    : Byte;
    BackGround    : Byte;
    RefX          : Integer;
    RefY          : Integer;
    AbsX          : Integer;
    AbsY          : Integer;
    Status        : Array[0..7] of Boolean;
    WindowNo      : Byte;
    AltFore       : Byte;
    AltBack       : Byte;
End
```

It is possible to define as many variables of the type PenInformationType as you want to. Typically it is advisable to define one variable for each task that is writing on the screen. This is necessary to ensure that no other tasks are changing the charactergenerator, color or pen position if they interrupt the task just after setting up the right parameters.

CharGen contains a pointer to a character generator. A character generator is an array of bitmaps, where each bitmap represents a character. Typically, the ASCII value for the character is used as an index within the character generator. The CharGen pointer is set up by use of the standard procedure SET-CHARACTERGENERATOR. The figure below shows an example of the character ”A” from a 6 x 8 character generator.

The colors on the screen are selected with the field variables **ForeGround and BackGround** in variables of type PenInformationType. The colors for a pen can be set by means of the standard procedure **SETCOLORS**. ForeGround and BackGround can take the following values:

16 different colors	(Only in PD5020, the different colors can be seen in the PD5000 manual. Black and White are used in the other controllers)
1 transparent color	
inverse writing-mode	

The difference between the foreground and the background writing is shown below for the character **A**.

The foreground color corresponds to the pixels defined to be ON in the bitmap specification (symbolised by 1 below) and the background color corresponds to the pixels defined to be OFF (symbolised by 0 below).

See also the previous chapter BITMAP.

0	**1**	**1**	**1**	0	0
1	0	0	0	**1**	0
1	0	0	0	**1**	0
1	0	0	0	**1**	0
1	**1**	**1**	**1**	**1**	0
1	0	0	0	**1**	0
1	0	0	0	**1**	0
0	0	0	0	0	0

If the foreground colour is set to transparent, only the background colour is written.

If the background colour is set to transparent, only the foreground colour is written.

If both the foreground colour and the background colour are set to transparent, nothing is written at all.

The 16 different colours are represented by 4 bit. When using the inverse writing mode, each of the 4 bits are inverted to give the resulting colour. (Only used in PD5020. Black and white can be inverted in the other controllers).

Writing to the screen is done with reference to a pixel position. This absolute pixel position is denoted by the field variables **AbsX** and **AbsY** in variables of type PenInformationType. This pixel position is not the same as cursor position and should not be interchanged with that.

The pen position X and Y can be assigned directly, or by the standard procedures **PenToAbs**, **PenTo**, **MovePen** and **PenRefTo** followed by the variable concerned. The variable must be of the type PenInformationType. If the variable is omitted, the variable called DefaultPen is used.

The standard procedures for changing the pen position and the corresponding results are listed below:

MyPen is used as a pen variable.

STANDARD PROCEDURE	RESULT
PenToAbs(MyPen,a,b)	MyPen.AbsX = a MyPen.AbsY = b
PenTo(MyPen,a,b)	MyPen.AbsX = MyPen.RefX + a MyPen.AbsY = MyPen.RefY + b
MovePen(MyPen,a,b)	MyPen.AbsX = MyPen.AbsX + a MyPen.AbsY = MyPen.AbsY + b

STANDARD PROCEDURE	RESULT
PenRefTo(MyPen,a,b)	MyPen.RefX = a MyPen.RefY = b MyPen.AbsX = a MyPen.AbsY = b

When writing on the screen, the AbsX for the pen is changed according to the current pen position.

Example:

```
VAR
  SmallChar : PenInformationType;

BEGIN

    SetCharactergenerator(SmallChar, SmallCharGenerator);
                (* select SmallCharactergenerator to be the charactergenerator when writing on
                the screen with the screeninformation variable called SmallChar *)

    SmallChar.ForeGround:= Black;     (* set foreground color black *)
    SmallChar.BackGround:= White;    (* set foreground color white *)

    PenToAbs(SmallChar,0,40);
                (* assign the pen position to AbsX=0 and AbsY=40 in the variable SmallChar *)

    Display(SmallChar,'This text is written with small characters');
                (* the text is written with characters from the charactergenerator
                SmallCharGenerator. The first character is positioned with the reference point
                in absolute pixel position 0,40 *)

    PenRefTo(1,1);
                (* assign the reference point to (1,1) and the absolute pen pos. to (1,1) for
                DefaultPen *)

    PenTo(MyPen,36,8);
                (* move the pen position in MyPen to position (36,8) relative to the reference
                point (MyPenRefX, MyPen.RefY) *)

    MovePen(0,8);
                (* move the pen position in DefaultPen relative to the absolute pen pos. for
                DefaultPen *)
```

The cursor is NOT used for writing at the screen, it is only used to point out variables at the screen when you want to change their value from the keyboard. So, the cursor is used for entering and changing data via the keyboard.

The symbol for the cursor is a bitmap and the bitmap is selected by the standard procedure **SetCursor**. Secondly, this procedure will display the cursor on the screen. Typically a cursor is only selected once in a program.

The size of the bitmap representing the cursor must not exceed the allocated memory space for the variable CursorHide. CursorHide is declared in the system file. The size of CursorHide must be at least the same size as the bitmap for the cursor.

If the colour graphic screen is used, CursorHide must be at least 4 times the size of the bitmap for the cursor. If the size of CursorHide is insufficient, an errorcode is generated.

Before the cursor is selected, the colours for the cursor bitmap must be set. The colors are set in the variable **ScreenInfo** at the field variables CursorForeGround and CursorBackGround. The specific colours depend on the selected cursor and the used display type. Furthermore the initial cursor position on the display must be selected. The field variables CursorX and CursorY must be set to the absolute pixel position for the cursor.

Three different standard procedures can be used to set or move the cursor to a specific position: **CursorToAbs**, **MoveCursor** and **CursorTo**.

The standard procedures for changing the cursor position and the corresponding results are listed below:

MyPen is used as a pen variable.

STANDARD PROCEDURE	RESULT
CursorToAbs(a,b)	ScreenInfo.CursorX = a ScreenInfo.CursorY = b
MoveCursor(a,b)	ScreenInfo.CursorX = ScreenInfo.CursorX + a ScreenInfo.CursorY = ScreenInfo.CursorY + b
CursorTo(MyPen,a,b)	ScreenInfo.CursorX = MyPen.RefX + a ScreenInfo.CursorY = MyPen.RefY + b

29 Keyboard.

The keyboards in the different Controllers are designed with a number of user-defined keys. The key functions depend upon the type of application, and may be defined by the Process-Pascal program.

Each key has its own keycode, starting with code 1 up to number of keys for that Controller.

The KeyBoardBuffer variable is a buffer of byte, where the operating system stores a key code when a key is pressed. It is also possible to achieve remote control, by - storing ”key codes” in the *KeyboardBuffer* via P-NET (an example of this can be found in VIGO for the PD 4000 Controller, by selecting the program called ‘Show PD4000 Controller’ from the right mouse menu).

The standard keyboard task is declared as a softwire interrupt task, connected to *KeyboardBuffer*, with interrupt condition ”*InternStore*" and ”*ExternStore*". The task will run each time a key is pressed, or a ”key code” is stored in the *KeyboardBuffer* via P-NET.

If one key is pressed, the number of that key is stored in *KeyboardBuffer* by the operating system. If the key is held down for more than 0.5 seconds, the operating system starts to send REPEAT codes with a frequency of 8 Hz. A repeat code consists of the key number + 128 ($80). If, when one key is held down, another key is also pressed, the code for the second key + 64 ($40) is stored in *KeyboardBuffer*.

Example:
Key number 4 is pressed. The code ***4*** is stored in *KeyboardBuffer*. Now the key is held down. After 0.5 seconds, the code ***132 ($84)*** is stored in *KeyboardBuffer* every 1/8 second. Now, with key number 4 still held down, key number 7 is pressed. The code ***71 ($47)*** is sent to the *KeyBoardbuffer*. If both keys are held down for 0.5 seconds, the code ***199 ($C7)*** is stored every 1/8 second. No release code is stored, when the keys are released.

Please refer to the specific Controller manuals for details about the number of keys available and the keycodes.

30 Real-time Clock and Calendar.

The CONTROLLER provides a real-time clock and calendar. The real-time clock and calendar is accessed through a system variable called DateTime, which is defined as SOFTWIRE 6 (in PD 5000 the real-time clock is found on Service.DataTime).

For PD 4000, DateTime is defined as an array of bytes in the system file.

For PD 5000, DateTime is defined as Record with eight fields, each defined as a byte in the system file.

Each byte or field in DateTime is defined below:

PD 5000	PD 4000	Description
Second	DateTime[0]	this byte holds the Seconds. The decimal range for seconds is from 0 to 59.
Minute	DateTime[1]	this byte holds the Minutes. The decimal range for minutes is from 0 to 59.
Hour	DateTime[2]	this byte holds the Hours. The decimal range for hours is from 0 to 23 in 24 hour mode or from 1 to 12 in 12 hour mode.
Day	DateTime[3]	this byte holds the Day of the Week. The range for day of the week is from 1 to 7, where Sunday = 1.
Date	DateTime[4]	this byte holds the Date. The range for date is from 1 to 28/29/30/31, depending on the month. The real-time clock provides automatic leap year.
Month	DateTime[5]	this byte holds the Month. The range for month is from 1 to 12, where Jan = 1 and Dec = 12.
Year	DateTime[6]	this byte holds the Years. The range for years is from 0 to 99.
Code	DateTime[7]	this byte holds a code for 12/24 hour operation and an indicator for AM-PM is available in 12 hour mode. The following values are used to select hour mode: DateTime[7]:=$00; (* 24 hour mode *) DateTime[7]:=$80; (* 12 hour mode, AM *) DateTime[7]:=$A0; (* 12 hour mode, PM *)

The 12/24 hour mode, indicated by Code, is not supported in PD 5000.

In 12 hour mode, DateTime[7] can be read to indicate if the time is AM or PM. It is recommended that DateTime[7] is set to one of the shown values in the program to ensure appropriate operation for the real-time clock and calendar.

The operating system in the Controller synchronises the DateTime with the real-time clock chip each time the Seconds change to 0. This can cause a problem when the time is set to 'just before midnight', 23:59:SS, the time does not reset to 00:00:00, but change to the previous time setting. When setting the time to 23:58:SS, the time will reset to 00:00:00 correctly.

31 Accessing not Declared Variables.

In larger systems with more controllers involved, you can have a need for accessing variables via P-NET which are not declared within the controller. The variables could be found in another controller or in a newly installed interface module connected to another part of the plant and therefore it may be unknown to a number of controllers.

In general it's not possible to access variables, e.g. to display a measured value, without having declared the variable first. However, the system variable called NodeList enables you to access not declared variables. Please refer to the SYS file for the controller in question if you want to see the SoftWire number for the variable.

Before you can access a variable, there are some basic elements, which must be known. The elements needed to access a variable are:

- P-NET node address
- SoftWire number or absolute address
- Offset
- Bit number

and of course, the type of the variable.

Besides the above, a variable can not be accessed correctly, unless the type of the module which holds the variable is known. According to the P-NET standard, you must specify if the module understands extended or complex P-NET addressing, addressing with offset and so on. Please refer to the chapter INTERFACE for further information.

All the above information for a variable can be gathered in a pointer by means of a NODELIST and a pointer function. The system variable NodeList is declared as an array of NodeListElement as shown in the following:

```
NodeList: ARRAY[1..10] of NodeListElement;
```

where each element is a record of the following type:

```
NodeListElement = RECORD
                    Code       : BYTE;
                    StdChannel : BOOLEAN;
                    DeviceType : INTEGER;
                    NodeAddr   : STRING[10];
                  END;
```

The user can change the number of elements in the NodeList. The size is declared to be 10 elements as default. The size of the NodeList can be changed to match the actual application.

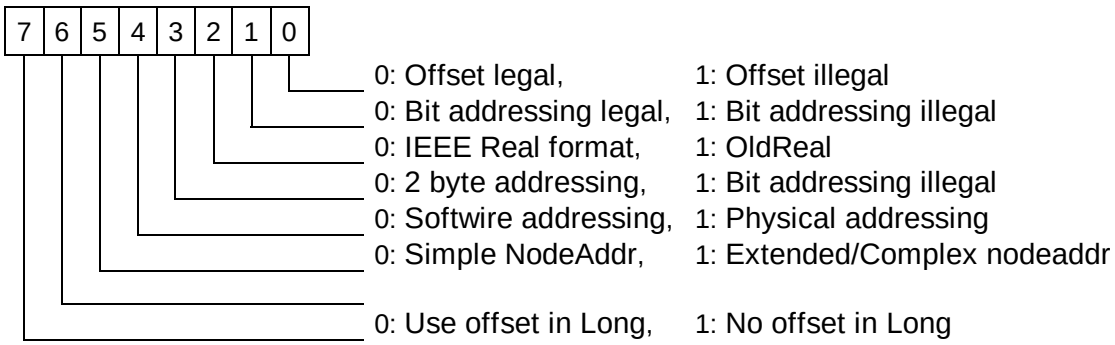
StdChannel must be set to TRUE if the device provides channels that follows the structure for general purpose channel types, as defined with the P-NET Standard.

DeviceType is an integer type denoting the device type for the module you want to access.

NodeAddr denotes the P-NET node address for the module you want to access. NodeAddr is declared as a string, where NodeAddr[0] holds the number of bytes that is needed to specify the entire node address including port numbers and node numbers.

The total length of the string denoting the node address must not exceed 25 characters. The port numbers and node numbers in the node address are NOT ASCII characters but hexadecimal numbers in the range 0..$7F, corresponding exactly to port numbers and node addresses.

The ***Code*** field indicates the capabilities for the node to access and is defined in the following way:


7 6 5 4 3 2 1 0
0: Offset legal, 1: Offset illegal
0: Bit addressing legal, 1: Bit addressing illegal
0: IEEE Real format, 1: OldReal
0: 2 byte addressing, 1: Bit addressing illegal
0: Softwire addressing, 1: Physical addressing
0: Simple NodeAddr, 1: Extended/Complex nodeaddr
0: Use offset in Long, 1: No offset in Long

Please refer to the P-NET standard for further details on the addressing facilities.

You may also refer to the VIGO Users Manual (Ref. No. 502 086), table Capabilities to get a list of values to use for *Code* for various device types (*Code* is the same as Capabilities in VIGO).

When you have defined an element in the nodelist, Code, DeviceType and NodeAddr, you can get access to the module specified within the list. You get access to the module by means of a pointer (see chapter POINTER TYPES how to declare and use a pointer) which is set to point to the variable you want to access.

The pointer you are using is set to point to the variable by means of the function POINTERTONODE, which is a standard function in Process-Pascal. PointerToNode operates on an element from the NodeList, a Softwire number, an offset and a bit number and it returns a pointer.

The syntax for PointerToNode is the following:

```
MyPtr -> PointerToNode(Node, SWNo [, Offset [, BitNo]]);
```

Node is an index to an element from the NodeList and SWNo specifies the actual Softwire number you want to access. If the variable is of a complex type, you can use the Offset to select a simple type variable. If the variable is a boolean array, you can use the BitNo to select the bit number. Offset and BitNo are optional and they are set to zero in the function call if omitted.

If Node = 0, i.e. the index is outside the NodeList array, then the pointer is set to an internal variable at the specified Softwire number. PointerToNode will also create a pointer to an internal variable if the NodeAddr field in the selected nodelist element is not specified (an empty string).

The PointerToNode is used both in the Service and Config programmes and in the example program for WHEN ERROR. You can find these files in the Process-Pascal library.

Example where PointerToNode is used:

```
VAR
    RealPtr : POINTER TO REAL;
    NodeNo, NodeSWNo, NodeOffset : INTEGER;

PROCEDURE ShowVariable;
BEGIN
    PenTo(0,0);
    Display('Select node number ');
    Update(NodeNo:2:0);
    PenTo(0,8);
    Display('Select softwire no ');
    Update(NodeSWNo:4:-3);
    PenTo(0,16);
    Display('Select offset      ');
    Update(NodeOffset:4:0);
    RealPtr -> PointerToNode(NodeNo, NodeSWNo, NodeOffset);
    PenTo(0,24);
    Display('Value for variable ');
    Display(RealPtr:6:2);
END;
```

32 PD GATEWAY.

PD Gateway is an advanced option in Process-Pascal, which enables you to integrate communication protocols for control and regulation equipment (PLC's etc.) from different manufacturers, directly into the Process-Pascal programme.

By means of the PD Gateway option, a "non P-NET compatible device" can be regarded as a P-NET interface unit and can be accessed as a "normal P-NET device" including error detection and error handling, from the entire P-NET system. On the other hand, the P-NET system can be regarded as a part of the "non P-NET system", just another unit, when you access it from the "non P-NET" side.

The "non P-NET system" is connected to the P-NET via a PD 5000 P-NET Controller, using either an RS-232 or RS-485 communication port. The PD 5000 P-NET Controller is connected to the "non P-NET system" via one of the physical ports, Port1 to Port3.

The principle is that instead of accessing the device directly following the P-NET standard, a Process-Pascal task is started from the operating system. This programme uses a virtual port, the Gateway port and a physical port to which the device is connected. The task is converting to and from the appropriate protocol for the actual device. When the answer from the device is ready, the Process-Pascal programme returns the answer to the operating system. The Process-Pascal task must be declared in as a SoftwireInterruptTask.

When the answer from the device is ready, the Process-Pascal programme returns the answer to the operating system in exactly the same manner as if the answer was received in a normal P-NET transmission.

The variables, which are to be accessed in the "non P-NET device" are declared in the master units where the access is required by the application, exactly as a normal variable declaration. The variables are declared as global variables with a NET address, including a net list and, depending on the communication protocol, an address specification. The net list for these variables must involve Port5 in the Gateway Controller to activate the Process-Pascal communication programme.

A GatewayRecord is declared in the system file for PD 5000, which is used to transfer data from the operating system to the Process-Pascal programme which interfaces to the "NON P-NET DEVICE" (CALL). After the Process-Pascal programme has performed the "NON P-NET" transmission, the same RECORD is used to transfer data from the Process-Pascal programme to the operating system (ANSWER).

The GatewayRecord is declared in the following way:

```
Record
   NodeAddress   : String[25]
   Control_Status : Byte;
   InfoLength    : Byte;
   Info          : Array[1..63] of Byte;
   Flags         : Array[0..7] of Boolean;
end
```

The fields *NodeAddress*, *Control_Status*, *InfoLength* and Info correspond to the same fields that are described in the P-NET standard. The operating system handles all NodeAddress conversion.

The task of the Process-Pascal program is to return Control/Status, InfoLength and Info according to the results of the NON P-NET transmission (or time consuming calculation). After this, the program must set the *GatewayDone* bit in the Flags field to True, (*Flags[7]:= True*) to activate the operating system, which then returns the answer to the P-NET master.

NodeAddress
This field is used only to transfer data from the operating system to the Process-Pascal programme (CALL). The length of the string indicates, how many numbers the string contains. If, for example, a variable is declared - AT NET:(01, 04, 02, 33) the length of the string will be 3, and the contents will be 04, 02, 33 (The first 01 indicates Port1, and is not transferred in the CALL).

Control_Status
This field is used to transfer data in CALL as well as ANSWER. First, it is used to transfer the instruction (as it is defined in the P-NET standard) in the CALL. In the ANSWER it is used to transfer instruction/status to the operating system - again according to the P-NET standard, thus enabling the Process-Pascal programme to return "Data Error" etc. to the operating system.

InfoLength
This field is also used both directions. It is used to tell, how many databytes the transmission is concerning. For example, if the transmission is a "LOAD 8 BYTE", datalength will be 8 in CALL as well as ANSWER. If the transmission is a "STORE 4 BYTE", datalength will be 4 in CALL, and 0 in ANSWER. So, the datalength is used to tell the Process-Pascal programme how many databytes are to be loaded/stored etc., and it is used to tell the operating system the number of databytes in the answer. The datalength must not exceed 56.

Info holds the entire info field, which consist of the internal address (Softwire no.), offset, bit number, data and error code as described below:

Addr
Addr is used only in CALL. Addr contains the internal address, as it is known from the P-NET standard. According to the P-NET standard, the address can be 2 or 4 bytes long, and it can be a Softwire number or a physical address. However, the address in Addr will allways be sign extended to 4 bytes. If the address in the P-NET block was 4 bytes, FLAGS[0] will be TRUE.

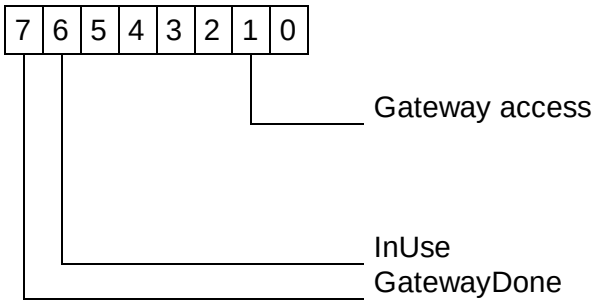
Offset
Offset is used only in CALL, according to the P-NET standard. If there was no offset in the P-NET block, the value 0 will be transferred in Offset. If there was an offset in the P-NET block, FLAGS[1] will be TRUE.

BitNo
BitNo is used only in CALL. If the transmission is a bit-transmission (indicated in the P-NET block by MSB in a 4 byte address SET), BitNo will contain the bit number, 0..7, and FLAGS[2] will be TRUE.

DATA
The Data field is used in CALL as well as ANSWER, and contains data as defined in the P-NET standard.

ErrorCode
This field is used in ANSWER only. If the "NON P-NET" transmission results in an error of some kind, the Process-Pascal programme can return an error code in this field. Refer to the chapter "WHEN ERROR" for information on, which error codes should be used. The operating system inserts 0 in this field in the CALL. If the ErrorCode is not 0 in the ANSWER, the operating system assumes there was a communication error, and does NOT use any data. Instead, the information "P-NET Error" is returned to the master.

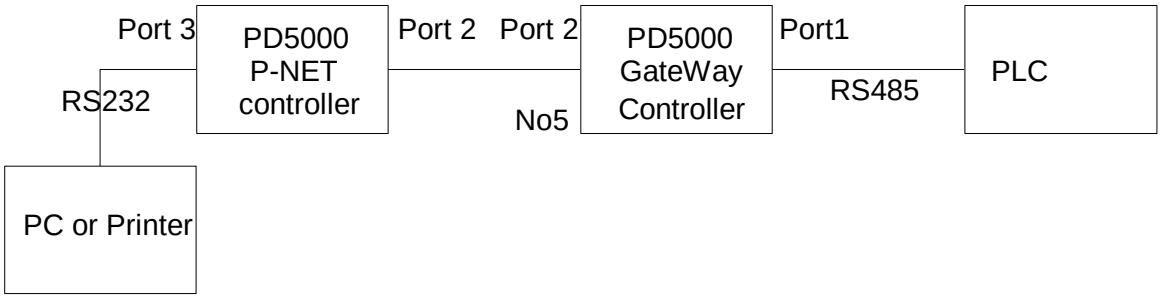
The *Flags* variable has the following meaning:



Before an interrupt is made to invoke the Process-Pascal program, *InUse* is set TRUE by the operating system. This is to prevent others from using the Gateway channel until this operation is Done.

When the Process-Pascal program has returned the response to *GatewayRecord* it must set *GatewayDone* to True. The Process-Pascal program should NEVER write to *InUse*. The operating system routines that are activated by the *GatewayDone* bit will clear *InUse*, when the operation is completed.

Example:



The PLC system consists of 2 PLC's, with node numbers 1 and 2.

The set up for the Gateway controller should be as follows:
Port_1.ActualMode.Protocol is set to *DatamodeInOut*,
Gateway.GatewayInterrupt is set to 5,
Port_1.ActualMode.BaudRate must be selected to match the PLC.
In the Gateway controller, the PLC-data are declared this way:

```
PLC1Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
```

```
                    AT NET:(5,1) SOFTWIRE:$1234;
PLC2Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                    AT NET:(5,2) SOFTWIRE:$1234;
```

In the P-NET controller, the PLC-data can be declared in 2 different ways. This way:

```
PLC1Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                    AT NET:(2,5,5,1) SOFTWIRE:$1234;
PLC2Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                    AT NET:(2,5,5,2) SOFTWIRE:$1234;
```

Or this way:

```
PLC1Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                    AT NET:(2,5) SOFTWIRE:aaaa;
PLC2Data: ARRAY[1..100] of INTEGER[DeviceType:5000]
                    AT NET:(2,5) SOFTWIRE:bbbb;
```

where aaaa and bbbb are the softwire numbers for PLC1Data and PLC2Data declared in the Gateway controller (from the Gateway controller MAP-file).

When a LOAD transmission of PLC1Data with index $31 is initiated from the P-NET controller, the Gateway controller returns "answer comes later" to the P-NET controller. Then the following data are transferred to the Info field in the GatewayRecord in the Gateway controller:

```
NodeAddress = $01
Instruction = $02                (LOAD)
DataLength  = $02                (INTEGER size)
Addr        = $00001234
Offset      = $0062              (index $31 * 2)
BitNo       NOT DEFINED
Data        NOT DEFINED
ErrorCode   = $0000
FLAGS       = 0/1/0/0/0/0/1/0
```

After inserting these data, the task with interrupt number 5 is started. This task must perform a transmission to the PLC, and insert the result in the Info field in the GatewayRecord:

```
NodeAddress  Don't care
Instruction = $82                (module error in the PLC)
DataLength  = $02
Address     Don't care
Offset      Don't care
BitNo       Don't care
Data        = 2233               (the data from PLC integer $31)
ErrorCode   = $0000              (no transmission errors occurred)
FLAGS       = 0/1/0/0/0/0/1/0
```

After inserting these data, the Process-Pascal program in the Gateway controller must insert TRUE in FLAGS[7]. This makes the operating system return the answer to the P-NET controller, with the data from the PLC.

Thus, the variables PLC1Data and PLC2Data can be accessed from the P-NET controller or from the Gateway controller, just as if they were variables inside a normal P-NET device.

33 Modules In Process-Pascal.

Modules are precompiled parts of a larger program with a precise external declaration part related to the host program. The external declaration part declares the constants, types, variables, procedures and functions that must be known by the module.

The idea about modules has certain advantages:

- The compiling time for a program is reduced because you must not compile the entire source code.
- You can include modules from a library without having the source code available.
- You can develop modules to hand over to a third part, without handing over the source code.
- You can divide large programs into smaller logically related modules.

Modules can consist of:

- Global constants
- Global types
- Global variables
- Global procedures
- Global functions
- A complete task
- A collection of the above.

The standard procedures RestartTask, StopTask, ContinueTask and Raise must not be used in modules. It is not allowed to declare external variables located at a net address in a module. It is not allowed to declare variables with interrupt in a module.

Example of a constant module:

```
MODULE;

CONST
  CH6x8 = ARRAY[$20..$A0] OF SMALLBITMAP[8] (
                [$20]:($06, $08, $00, $00, $00, $00, $00, $00, $00, $00) (*space *)
                [$21]:($06, $08, $20, $20, $20, $20, $00, $00, $20, $00) (* ! *)
                                        . . . . . . . . . . . . . . .
                                                                        );
END.
```

The above module is very simple and no external information is required by the module.

Compiling a module is done exactly like compiling a program. The compiler is called and the source file is stated. Then the compiler generates a LST file and a COD file. No MAP file or SMB file is generated for modules.

When compiling the module, the declarations in the external part are assumed to be the right identifiers and the references from these identifiers are used within the compiled code.

When the module is imported in a host program, the references to the external declarations are substituted with the appropriate references from the global declarations in the host program. If the declarations in the host program are missing or differ from the declarations in the external part, a compiling error is generated.

Example of a procedure/function module:

```
MODULE;
EXTERNAL
BEGIN
    CONST
      constant1 = REAL;
    VAR
       variable1 : INTEGER;
       variable2 : REAL;
END;

PROCEDURE ModulProcedure(VAR p1: LONGREAL);
BEGIN
    variable1:=variable2 * constant1;
    p1:=variable2;
END;

FUNCTION ModulFunction(VAR p1: LONGREAL):BOOLEAN;
BEGIN
    variable1:=variable2 * constant1;
    p1:=variable2;
    ModulFunction:=(p1=variable1);
END;
END.
```

The procedure and function in the above module operates on various external parameters. These external parameters must be declared in an EXTERNAL part exactly as they are found in the host program.

Example of a task module:

```
MODULE;
EXTERNAL
BEGIN
    VAR
       Alarm : BOOLEAN;
    PROCEDURE LedOnOff(b:BOOLEAN);
END;

Task FlashWithLED TIMEDINTERRUPT:0.5;
VAR
    Led : BOOLEAN;
BEGIN
    Led:=OFF;
    LOOP
        IF Alarm THEN
        BEGIN
            LedOnOFF(Led);
            Led:=NOT Led;
        END
        ELSE
            LedOnOff(ON);
        Changetask;
    END;
END;
END.
```

The compiled module is inserted in a host program by means of an IMPORT statement. The IMPORT statement includes the entire module in the host program at the point where it is written. The IMPORT statement has the following syntax:

```
IMPORT <filename>
```

The <filename> must specify drive, directory and the complete name for the module code file.

Example:

```
IMPORT CH6x8.COD;
```

34 Process-Pascal Reference Lookup.

The following list describes all the procedures and functions in Process-Pascal which are extensions to ISO 7185 STANDARD PASCAL.

[ ] denotes that the enclosed parameter is optional. If not used, the compiler inserts a default value.

34.1 Standard procedures.

34.1.1 ALARMHORNONOFF

```
AlarmHornOnOff(b);
```

This procedure can only be used on a PD5000 controller. The alarm output on the PD5000 is set or reset with this procedure. b is a boolean. If b is true, the output is set. By calling this procedure, any preceding pulsing with the alarm output is stopped (see below).

Example:

```
AlarmHornOnOff(ON);
   (* The alarm output is set  *)
AlarmHornOnOff(OFF);
   (* The alarm output is reset  *)
```

34.1.2 ALARMPULSEON

```
AlarmPulseOn(OnTime, OffTime);
```

This procedure can only be used on a PD5000 controller. By calling this procedure, the alarm output on the PD5000 is toggled with the specified on and off time. The OnTime and OffTime are assigned as real values. The resolution for the time is 1/128 second (same resolution as a timer).

Example:

```
AlarmPulseOn(0.5, 0.5);
   (* The alarm output is switched each half second *)
```

34.1.3 AND

```
And(variable, expression);
```

This procedure performs a logical AND instruction directly on the variable with the expression parameter. The variable can be declared as internal or external.

Example:

```
And(FlagReg, $55);
   (* The FlagReg variable is And'ed with $55 *)
```

34.1.4 BOX

```
Box([peninfo,] SizeX, SizeY);
```

This procedure draws a box figure as a rectangle. The box is drawn with FOREGROUND COLOR for the specified pen, starting in the current pen position. The size of the box is specified in SizeX and SizeY. When the procedure is done, the pen position is moved SizeX pixels in the X-direction and SizeY pixels in the Y-direction.

Example:

```
Box(MyPen, 25, 50);
   (* A box is drawn on the screen with a size of 25 by 50 pixels *)
```

34.1.5 BOXTO

```
BoxTo([peninfo,] PosX, PosY);
```

This procedure draws a box figure as a rectangle from the current pen position to (PosX,PosY). PosX and PosY are relative to the upper left corner of the window. The box is drawn with FOREGROUND COLOR for the specified pen. The size of the box is determined by the current pen position and PosX and PosY. The pen position is not changed.

Example:

```
BoxTo(MyPen, 150, 240);
(* A box is drawn on the screen starting at the current pen position and ending at
position (150,240) relative to the upper left corner of the window *)
```

34.1.6 CHANGETASK

```
ChangeTask;
```

ChangeTask is a procedure without parameters and it is used to change the program execution to an other task. ChangeTask can be used in all types of task.

Example:

```
LOOP              (* LOOP forever   *)
   ..             (*              *)
   ..
   ChangeTask;    (* Change to another task  *)
END;
```

34.1.7 CLEAR

```
Clear(Error [,errorbit, .., errorbit]);
```

Clear is used to clear error bits, generated by the automatic error detection system (see also WHEN ERROR, DISABLE(ERROR), ENABLE(ERROR)). If the bit specification is omitted, all error bits are cleared, otherwise the specified error bits are cleared.

The different error bits are:

```
PnetError,          HisError,ModuleError,    ActError,DataError,
BufferError,                                 ArithmicError,
IndexError,                                  ConvertError
```

Example:

```
Clear(Error);
   (* clear all error bits *)
Clear(Error, BufferError, IndexError);
   (* clear buffer error and index error bits *)
```

34.1.8 CLEARWINDOW

```
ClearWindow(peninfo);
```

ClearWindow is used to clear the window specified by peninfo. The entire window is cleared to background color specified by peninfo. The pen position is not affected by calling the ClearWindow procedure.

Example:

```
ClearWindow(AlarmPen);
   (* clear the window specified by AlarmPen *)
```

34.1.9 CLOSEWINDOW

```
CloseWindow[(peninfo)];
```

The procedure can only be used for PD5020. CloseWindow is used to close a window , where the window number is specified by peninfo. If the window is closed already, the procedure does nothing. If any other windows are opened within the specific window, all these windows are closed first. The window is closed by drawing the corresponding background colours for the other windows within the field that this window is covering.

Example:

```
CloseWindow(MyPen);
   (* close the window specified by MyPen *)
```

34.1.10 CONTINUETASK

```
ContinueTask(TaskIdentifier);
```

A SUSPENDED task can change to READY status, if another RUNNING task calls the standard procedure CONTINUETASK with the appropriate task identifier, ContinueTask(TaskIdentifier). This will insert the task in the right task chain and make the task "TaskIdentifier" continue from where it was last stopped or interrupted.

Example:

```
ContinueTask(WeightBatching);
```

34.1.11 CONTRASTCONTROL

```
ContrastControl(w);
```

The procedure can only be used for PD5010 and PD5015. ContrastControl is used to set or change the contrast on the screen. The parameter w is an absolute value, that is passed to the screens contrast register. The range for the contrast is from 0 to $F.

Example:

```
ContrastControl(0);
    (*Contrast is set to no contrast *)
ContrastControl($F);
    (*Contrast is set to full contrast *)
ContrastControl(ContrastValue+1);
    (*Contrast is set one up corresponding to the actual value *)
```

34.1.12 CURSORTO

```
CursorTo([peninfo], x, y);
```

The procedure moves the cursor position (x,y) relative to the reference point for peninfo. If peninfo is omitted, then the pen variable within the block called Default-Pen, is used. The cursor is automatically removed from the old position and displayed at the new position. If a cursor not is selected with the SetCursor procedure, an error is generated.

Example:

```
CursorTo(MyPen, CursorStepX, CursorStepY);
(* The cursor is moved to position (CursorStepX, CursorStepY) relative to
the reference point for MyPen *).
```

34.1.13 CURSORTOABS

```
CursorToAbs(x, y);
```

The procedure sets the absolute cursor position to (x,y). The cursor is automatically removed from the old position and displayed at the new position. If a cursor not is selected with the SetCursor procedure, an error is generated.

Example:

```
CursorToAbs(CursorOffsetX, CursorOffsetY);
    (* The cursor is moved to position (CursorOffsetX, CursorOffsetY) *).
```

34.1.14 CYCLICTASK

```
CyclicTask;
```

A TIMEDINTERRUPT task and a SOFTWIREINTERRUPT task can change to a CYCLIC task by means of the standard procedure CYCLICTASK. Changing the task type, will insert the task in the cyclic sequence and the program execution for the task will continue until it meets a ChangeTask statement.

Example:

```
Task FlowControl TimedInterrupt:1.0;
(* the task is declared as a timed interrupt task *)
BEGIN
  . . .
  CyclicTask;
(* change the task type to a cyclic task *)
  . . .
END;
```

34.1.15 DISABLE

```
Disable(Error [,errorbit, .., errorbit]);
```
or
```
Disable(TimedInterrupt);
```
or
```
Disable(SoftwireInterrupt);
```
or
```
Disable(Interrupt);
```
or
```
Disable(ChangeTask);
```

Disable(Error)
Disable is used to disable errors, generated by the automatic error detection system (see also WHEN ERROR, CLEAR(ERROR), ENABLE(ERROR)). If the bit specification is omitted, all errors are disabled, otherwise the specified errors are disabled.

The different errors to disable are:

```
PnetError,    HisError,ModuleError,  ActError,DataError,
BufferError,  ArithmicError,         IndexError,        ConvertError,
PnetReport,   ModuleReport,          DataReport
```

Example:

```
Disable(Error);
(*disable all errors, i.e. disable the entire automatic error detection system*)
```

```
Disable(Error, ModuleError, DataError);
(* disable module and data errors detected during P-NET transmission *)
```

Disable(TimedInterrupt)
Disable(TimedInterrupt) is used in cyclic task, to prevent timed interrupt task to interrupt. All timed interrupt task are disabled by this procedure. Disabling the timed interrupt tasks will not change the status of these tasks. This means that they are not removed from the task chain, and when the timed interrupt tasks are enabled again, the timed interrupt tasks will try to catch up with the lost time (if any).

If timed interrupt is disabled in a procedure or in a function, the interrupt status is automatically set back to the state it was before the call when the procedure or function is exited.

Disable(SoftwireInterrupt)
Disable(SoftwireInterrupt) is used in cyclic task, to prevent softwire interrupt task to interrupt. All softwire interrupt task are disabled by this procedure. Disabling the softwire interrupt tasks will not change the status of these tasks.

If softwire interrupt is disabled in a procedure or in a function, the interrupt status is automatically set back to the state it was before the call when the procedure or function is exited.

Disable(Interrupt)
Disable(ChangeTask)
Disable(Interrupt) and Disable(ChangeTask) will disable both TimedInterrupt and SoftwireInterrupt task (see description above).

34.1.16 DISPLAY

```
Display([peninfo,] information: size [:format]);
```

Display is used to show a bitmap, writing text or displaying the value of a variable, an expression or a function on the screen. The bitmap, text or variable is shown with the reference point for the first character in (peninfo.x, peninfo.y). When the procedure is done, peninfo.x has been moved to the right ((size * (width of one digit) for numerals, and after the last character or the number of pixels specified by FORMAT for strings), i.e peninfo.x is pointing at the first pixel after the field. If peninfo is omitted, then the pen variable within the block called DefaultPen, is used.

INFORMATION must be of simple type or string or bitmap. The Information can be declared as internal in the controller, or it can be declared to be located on the P-NET network.

If the information is a **string type**, the parameter SIZE is optional and denotes the maximum **field width**, in pixels, for representing the string on the screen. If the field width is larger than the actual string, the remaining field is filled with blank pixels (background colour), otherwise the string is written until the maximum field width is exceeded. If omitted, the string is written with the actual number of characters. When the procedure is done, PenInfo.X has been moved SIZE pixels to the right, or to the pixel after the last written character if SIZE is omitted.

If the information is a type different from string, SIZE denotes the **number of characters** that is written for the variable. The parameter FORMAT is a value for representing the information on the screen.

If information is an expression, a variable or result of type TIMER, REAL or LONGREAL, format has the following meaning:

0-..	Number of digits to the right of the decimal point. Default value is 2.
-1	The variable is represented with floating-point.
-2	The variable is represented with exponent. For the type TIMER or REAL the exponent is always 2 digits and a sign. For the type LONGREAL the exponent is always 3 digits and a sign.

If *information* is a variable or result of a type BYTE, WORD INTEGER or BOOLEAN, format has the following meaning:

0	Decimal representation. This is the default value.
-3	Hexadecimal representation.
-4	Binary representation.
-5	Decimal representation with leading zeros.

If *information* is a variable or result of type char or byte, format additionally has the following meaning:

-6	ASCII representation.

If *information* is of type bitmap or string, format is not used.

Example:

```
(* b is a BYTE, r is a REAL *)
(* gives the following representation:  *)

b:=255;
Display(b:3:0);   (* 255 *)
Display(b:4:-3);  (* 00FF *)
Display(b:8:-4);  (* 11111111 *)

r:=12.345678;
Display(r:7:2);   (*   12,34 *)
Display(r:7:-1);  (* 12,3456 *)
Display(r:7:-2):  (* 1,2e+01 *)

Display('Process-Pascal');     (* Process-Pascal *)
Display('Process-Pascal':60);  (* Process-Pa *)
                  (* each character is 6 pixels wide *)

Display(LargeChar,'Process-Pascal');
Display(InputString);
Display(ValveSymbol);
```

34.1.17 DISPLAYONOFF

```
DisplayOnOff(b);
```

The procedure can switch the screen on or off. b is a boolean.
Example:

```
DisplayOnOff(ON);    (*The screen will turn on. *)
```

34.1.18 ENABLE

```
Enable(Error [,errorbit, .., errorbit]);
```

or

```
Enable(TimedInterrupt);
```

or

```
Enable(SoftwireInterrupt);
```

or

```
Enable(Interrupt);
```

or

```
Enable(ChangeTask);
```

Enable(Error)
Enable is used to Enable errors, to be generated by the automatic error detection system (see also WHEN ERROR, CLEAR(ERROR), DISABLE(ERROR)). If the bit specification is omitted, all errors are enabled, otherwise the specified errors are enabled.

The different errors to enable are:

```
PnetError,    HisError,ModuleError,  ActError,DataError,
BufferError,  ArithmicError,         IndexError,        ConvertError,
PnetReport,   ModuleReport,          DataReport
```

Example:

```
Enable(Error);
(*enable all errors, i.e. enable the entire automatic error detection system
*)
```

```
Enable(Error, PnetError, PnetReport);
(* enable P-NET errors detected during P-NET transmission and produce
a report, i.e. the operating system stores an element in the
InterFaceErrorBuffer *)
```

Enable(TimedInterrupt)
Enable(TimedInterrupt) is a standard procedure to use in cyclic task, to enable timed interrupt task to interrupt. All timed interrupt task are enabled by this procedure. Timed interrupt task which ought to have run, will execute after enabling and try to catch up with the lost time (if any). Timed interrupt task are enabled as default in cyclic tasks.

If timed interrupt is enabled in a procedure or in a function, the interrupt status is automatically set back to the state it was before the call when the procedure or function is exited.

Enable(SoftwireInterrupt)
Enable(SoftwireInterrupt) is a standard procedure to use in cyclic task, to enable softwire interrupt task to interrupt. All softwire interrupt task are enabled by this procedure. Softwire interrupt task which ought to have run, will execute in the priority order. Softwire interrupt task are enabled as default in cyclic tasks.

If softwire interrupt is enabled in a procedure or in a function, the interrupt status is automatically set back to the state it was before the call when the procedure or function is exited.

Enable(Interrupt)
Enable(ChangeTask)
Enable(Interrupt) and Enable(ChangeTask) will enable both Timed- and SoftwireInterrupt task (see description above).

34.1.19 INITBUFFER

```
InitBuffer(buffername);
```

Buffers must always be initiated before they are used the first time. This is done with the standard procedure InitBuffer(buffername).

Example:

```
InitBuffer(KeyboardBuffer);
```

34.1.20 INITPORT(n)

```
InitPort(n);
```

The parameter settings for a communication port (port number n) in a controller are assigned to a system variable. When any of the parameters for a communication port are changed, the corresponding procedure InitPort(n) must be called to reconfigure and initialise the communication port with the new data from system variable. See the manual for the controller in question to get a list of available ports and the possible settings. Please note that the procedure is called *InitPort1*, and is without any parameters when using a PD 4000 controller.

Example:

```
PD 5000:
        Port_2.ActualMode.PnetNo:=NewNodeNo;
        InitPort(2);  (* change the P-NET number for port2 and initialise the port*)

PD 4000:
        ModePort1.NoOfMaster:=7;
        InitPort1;    (* change number of masters to 7 and initialise port1 *)
```

34.1.21 INTERRUPTTASK

```
InterruptTask;
```

A CYCLIC task and a TIMEDINTERRUPT task can change to a SOFTWIREINTERRUPT task, if it was originally declared as a softwireinterrupt task, by means of the standard procedure INTERRUPTTASK. The interrupt connection is set to the initial softwireinterrupt number (declared in the task head). The task continues in the next statement.

Example:

```
Task CalculateTotals SoftwireInterrupt:12;
(* the task is declared as a softwire interrupt task *)
BEGIN
  . . .
    CyclicTask;       (*change task type and perform the calculations *)
(* calculate totals *)

    InterruptTask;   (*change the task back to a softwire interrupt task *)
      . . .
END;
```

34.1.22 LEDONOFF

```
LedOnOff(b);
```

The procedure can switch the LED on the display unit on or off. b is a boolean. If b is true the LED is on. The procedure can only be used for PD5010 and PD5015.

Example:

```
LedOnOff(ON);       (* The LED will go on *)
```

34.1.23 LIGHTCONTROL

```
LightControl(w);
```

LightControl is used to set or change the light on the screen. The parameter w is an absolute value, that is passed to the light register for the screen. The range for the lightvalue is from 0 to 7. The procedure can only be used for PD5010 and PD5015.

Example:

```
LightControl(0);                (*Light is set to no light *)
LightControl(7);                (*Light is set to full light *)
LightControl(LightValue+1);     (*Light is increased *)
```

34.1.24 LIGHTONOFF

```
LightOnOff(b);
```

LightOnOff is used to switch the light on or off at the screen. The procedure can only be used for PD5010 and PD5015.

Example:

```
LightOnOff(ON);  (*       Light is switched on *)
```

34.1.25 LINE

```
Line([peninfo,] OffsetX, OffsetY);
```

This procedure draws a line to a point that is a relative distance from the current pen position. The line is drawn with FOREGROUND COLOR for the specified pen, starting in the current pen position. The ending point for the line is specified by OffsetX and OffsetY as a relative distance from the current pen position. The thickness of the line is 1 pixel. When the procedure is done, the pen position is moved OffsetX pixels in the X-direction and OffsetY pixels in the Y-direction. The point at the final pen position is not drawn.

Example:

```
Line(MyPen, 50, 0);
   (* A vertical line is drawn on the screen with a length of 50 pixels *)
```

34.1.26 LINETO

```
LineTo([peninfo,] PosX, PosY);
```

This procedure draws a line from the current pen position to position (PosX,PosY). PosX and PosY are relative to the upper left corner of the window. The line is drawn with FOREGROUND COLOR for the specified pen. The thickness of the line is 1 pixel. The point at position (PosX,PosY) is not drawn. The pen position is not changed.

Example:

```
LineTo(MyPen, 250, 40);
```

```
    (* A line is drawn on the screen starting at the current pen position
    and ending at position (250,40) relative to the upper left corner of the
    window. *)
```

34.1.27 MAXRUNTIME

```
MaxRunTime(time);
```

The MAXRUNTIME for a task is initially given by a real constant and is declared in seconds and the resolution is 1/128 second. The default MAXRUNTIME is 300 seconds.

The Max Runtime can be changed during program execution with the standard procedure MAXRUNTIME(time), where time must be a constant or a variable, denoting the new max runtime in seconds.

Examples:

```
MaxRunTime(NewRunTime);

MaxRunTime(0.3);
```

34.1.28 MOVECURSOR

```
MoveCursor(x, y);
```

The procedure moves the cursor position (x,y) relative to the absolute cursor position. The cursor is automatically removed from the old position and displayed at the new position. If a cursor is not previously selected with the SetCursor procedure, an error is generated.

Example:

```
MoveCursor(CursorStepX, CursorStepY);
    (* The cursor is moved CursorStepX pixels in the x-direction and
    CursorStepY pixels in the y-direction *).
```

34.1.29 MOVEPEN

```
MovePen([peninfo,] x, y);
```

The procedure moves the absolute pen position for peninfo to position (x,y) relative to the absolute pen position. If peninfo is omitted, then the pen variable within the block called DefaultPen, is used.

Example:

```
MovePen(36, 16);
    (* The absolute pen position for DefaultPen is moved 36 pixels in the
    x-direction and 16 pixels in the y-direction *).
```

34.1.30 MYSWNO

```
MySWNo(identifier, SoftWireNo, VarOffset);
```

This procedure finds the SoftWire number for a global constant or variable and returns this number in SoftWireNo. If the identifier denotes a field in a complex variable, the actual offset for this field, in bytes is returned in VarOffset. SoftWireNo and VarOffset must both be integer type variables.

Examples:

```
MySWNo(Recipe[Last].Stirring, SWNo, VarOffset);

MySWNo(DigitalModule, SWNo, VarOffset);
```

34.1.31 OPENWINDOW

```
OpenWindow([peninfo,] WindowWidth, WindowHeight);
```

OpenWindow is used to define and open a new window. The window can be opened anywhere on the screen with any size within the screen. There is no limit in the numbers of windows which can be defined, but a maximum of 16 windows (ranging from 0 to 15) can be open at the same time. If the window is opened already, the procedure does nothing. If too many windows are opened, an error code is generated.

The upper left corner of the window is located at the current pen position when the procedure is called. The size of the window is specified by WindowWidth and WindowHeight. The size of the window includes the frame. The window is drawn with the background colour and the frame for the window is drawn with the foreground colour, all specified by peninfo. The thickness of the window frame is 2 pixels for the vertical lines and 3 pixels for the horizontal lines (default values). The thickness of the frame can be changed by the SetWindowFrame procedure.

The window number is dynamically assigned by the operating system. When opening the window, the "parent" window number must be specified in the pen, in the field PenInfo.WindowNo. When the procedure is done, the new window number is stored in peninfo. This window number must be found in the pen when writing in the window. It is very important to keep order in the window numbers. If the "parent" window is closed, all the "child" windows are automatically closed as well. To ensure an effective program execution it is advisable to use a status boolean indicating whether the window is open or not. If the window is not open, the program part for the window should not be executed.

The current pen position is not affected by this procedure. The procedure can only be used for PD5020.

Example:

```
OpenWindow(MyPen,200,150);
   (* open a window with size 200 by 150 pixels *)
```

34.1.32 OR

```
Or(variable, expression);
```

This procedure performs a logical OR instruction directly on the variable with the expression parameter. The variable can be declared as internal or external.

Example:

```
Or(FlagReg, $80);
   (* The FlagReg variable is Or'ed with $80 *)
```

34.1.33 PENREFTO

```
PenRefTo([peninfo,] x, y);
```

The procedure sets the reference point and the absolute pen position for peninfo to position (x,y). If peninfo is omitted, then the pen variable within the block called DefaultPen, is used.

Example:

```
PenRefTo(MyPen, 0, 0);
   (* The reference point and the absolute pen position for MyPen is set
   to the top left corner of the screen *).
```

34.1.34 PENTO

```
PenTo([peninfo,] x, y);
```

The procedure moves the absolute pen position for peninfo to position (x,y) relative to the reference point. If peninfo is omitted, then the pen variable within the block called DefaultPen, is used.

Example:

```
PenTo(60, 8);
   (* The absolute pen position for DefaultPen is moved 60 pixels in the
   x-direction and 8 pixels in the y-direction, relative to the reference
   point *).
```

34.1.35 PENTOABS

```
PenToAbs([peninfo,] x, y);
```

The procedure sets the absolute pen position for peninfo to position (x,y). If peninfo is omitted, then the pen variable within the block called DefaultPen, is used. The reference point is not changed.

Example:

```
PenToAbs(20, 32);
   (* The absolute pen position for DefaultPen is set to (20,32) *).
```

34.1.36 PERFORMUPDATE

```
PerformUpdate;
```

The procedure is used to pass a value from the inputstring to a variable. The procedure converts the digits in the inputstring and stores these data in the variable in the right format. The procedure will only pass the value to the variable if the cursor is pointing at the field for the variable and if the variable is displayed at the screen with the procedure UPDATE. If the cursor is not pointing at a field for a variable, the procedure does nothing.

Example (keyboard task):

```
$0F: Performupdate;
```

34.1.37 RAISE

```
Raise([TaskIdentifier,] Error [,errorbit, .., errorbit]);
```

Raise is used to force an error state, ignoring the automatic error detection system (see also WHEN ERROR, CLEAR(ERROR), DISABLE(ERROR) and ENABLE(ERROR)). If the bit specification is omitted, all errors are raised, otherwise the specified errors are raised. An error can be raised in a specific task denoted by TaskIdentifier, or the error is raised within the task which called the procedure.

The different errors to raise are:

```
PnetError,    HisError,ModuleError,  ActError,DataError,
BufferError,  ArithmicError,         IndexError,        ConvertError,
```

Examples:

```
Raise(CommTask, Error);
(* raise all errors in the communication task, i.e. force an error state and
   move program execution to the latest WHEN ERROR part next time
   the task is running *)
```

```
Raise(Error, PnetError);
(* raise P-NET error, i.e. force an error state and move program execu-
   tion to the last WHEN ERROR part *)
```

34.1.38 RESTARTTASK

```
RestartTask;
```

A RUNNING task can force itself to RESTART from the beginning of the task. To perform a restart for a task, the standard procedure RESTARTTASK is called. After calling RESTARTTASK, the program execution will continue with the first statement within the task.

Example:

```
RestartTask;
```

34.1.39 RETRYIFLEGAL

```
RetryIfLegal;
```

In a situation where the program has detected a transmission error and the program execution has been moved to the WHEN ERROR part, the program can retry the P-code that caused the error. To do so, a standard procedure RetryIfLegal must be called.

WARNING: When using RetryIfLegal, the program execution retries the P-code in which the error occurred and there is a risk of an infinite loop or a very slow system in case of many errors. If using the RetryIfLegal procedure, you should always implement a counter and a maximum value for the counter to avoid that your program locks. The RetryIfLegal procedure can only be executed if the "WHEN ERROR program" was invoked by a transmission error.

34.1.40 RETURN

```
Return;
```

The procedure is used to return program execution from a WHEN ERROR part. The program execution returns to the statement which caused the error and continues in the P-code right after. The procedure can only be called in a WHEN ERROR THEN program part.
WARNING: When using RETURN, there is a risk of erroneous data in the succeeding calculations.

Example:

```
When Error THEN
BEGIN
   i:=i+1;
   IF i > MaxTries THEN Return;
END;
```

34.1.41 SETCHARACTERGENERATOR

```
SetCharacterGenerator(peninfo, CharGen);
```

The procedure inserts the charactergenerator CharGen in the chosen peninfo. All the following Display and Update procedures with this peninfo will use the charac-tergenerator CharGen.

Example:

```
SetCharacterGenerator(DefaultPen, CharGen);
SetCharacterGenerator(LargeCharPen, LargeCharGen);
```

34.1.42 SETCOLORS

```
SetColors(peninfo, foreground, background);
```

The procedure sets the foreground and background colors in the specified pen.

Example:

```
SetColors(MyPen, Inverse, Transparent);
```

34.1.43 SETCURSOR

```
SetCursor(CursorRef);
```

The procedure selects CursorRef to be the actual cursor.
CursorRef can be defined as a LARGEBITMAP where ReferenceX and ReferenceY denotes an offset from the upper left corner of the cursor to a referencepoint, which is the point that must be inside the field on the screen where a variable is to be updated. If CursorRef is defined as a SMALLBITMAP, the upper left corner of the cursor is used as referencepoint (0,0). Before SetCursor is called, the colours and the position on the screen must be selected. Calling SetCursor will automatically display the cursor according to the previous settings. If a cursor has already been selected before with SetCursor, the old cursor is removed from the screen before the new cursor is displayed.

Example:

```
SetCursor(BlackCursor);
```

34.1.44 SETCURSORCOLORS

```
SetCursorColors(foreground, background);
```

The procedure sets the foreground and background colours for the cursor. The cursor is displayed on the screen with the new colours. If a cursor is not previously selected with the SetCursor procedure, an error is generated.

Example:

```
SetCursorColors(Red, Green);
```

34.1.45 SETCURSORTYPE

```
SetCursorType(peninfo, cursortype);
```

The procedure selects a cursor type. The cursor can be a cursor of bitmap type (CursorType = 0) or the cursor can be a of reticule type (CursorType = 1). A cursor of the bitmap type is selected as default. The procedure can only be used for PD5020.

Example:

```
SetCursorType(DrawPen, 1);
```

34.1.46 SETINPUTSTRING

```
SetInputString(str);
```

The procedure selects the string str to be the actual inputstring. All the following Update procedures will use the inputstring str.

Example:

```
SetInputString(InputString);
```

34.1.47 SETSCREEN

```
SetScreen(VideoScreen);
```

The procedure selects the variable VideoScreen of type VIDEOBITMAP to be the screen.

Example:

```
SetScreen(Screen);
```

34.1.48 SETVIDEO

```
SetVideo(peninfo, Width_of_screen, Height_of_screen);
```

The procedure clears the screen by setting the entire videoram to background colour. Furthermore it passes the width and height for the screen to ScreenInfo.Width and ScreenInfo.Height in the video controller. The cursor is automatically displayed on the screen again after clearing.

It is recommended to use this procedure to clear the screen when selecting between various screen layouts, because this procedure will clear information in the operating system on valid cursor positions within an update field on the previous display. This will prevent that you can update data belonging to a previous display after selecting a new display. In the PD5020 controller all windows are automatically closed.

Example:

```
SetVideo(DefaultPen, ScreenWidth, ScreenHeight);
```

34.1.49 SETWINDOW

```
SetWindow(ScreenX, ScreenY);
```

The procedure passes ScreenX and ScreenY to ScreenInfo.ScreenX and ScreenInfo.ScreenY in the video controller as an offset to the startaddress for the videoram to select a window. The parameters ScreenX and ScreenY is the position for the pixel at the top left corner on the screen. SetWindow can be used to scroll through the videoram.

Example:

```
SetWindow(0, 0);
```

34.1.50 SETWINDOWFRAME

```
SetWindowFrame(peninfo, topborder, rightborder, bottomborder,
               leftborder);
```

The procedure selects the thickness of the frame for a window. The window number must be found in PenInfo. The thickness of the frame can be specified for all 4 borders, starting with the top border and the following in clockwise order. The default values for the thickness of the border is set to 2 pixels for the vertical and 3 pixels for the horizontal. The procedure can only be used for PD5020.

Example:

```
SetWindowFrame(EditPen, 4 ,6 ,4 ,6);
```

34.1.51 STOPTASK

```
StopTask(TaskIdentifier);
```

A READY task can change to SUSPENDED status, if another RUNNING task calls the standard procedure STOPTASK with the appropriate task identifier, StopTask(TaskIdentifier). This will prevent the task "TaskIdentifier" from running any more, until it is changed to READY again from another task by means of CONTINUETASK(TaskIdentifier) statement. When a task is SUSPENDED, i.e. stopped or has come to an END for the task, the task is removed from the task chain. This means that a timed interrupt task will not try to catch up with the lost interrupts when starting again after it has been stopped.

Example:

```
StopTask(WeightBatching);
```

34.1.52 TIMEDINTERRUPTTIME

```
TimedInterruptTime(time);
```

The time interval for running a TimedInterrupt task can be changed during program execution by means of the standard procedure TIMEDINTERRUPTTIME(time), where time can be a constant or a variable, denoting the interval time in seconds. The time is specific for the task which calls the procedure so the procedure must be called from the timed interrupt task where the time must be changed. If the procedure is called from a cyclic task or a softwire interrupt task, it has no effect.

Example:

```
TimedInterruptTime(FastCheckTime);
```

34.1.53 TIMEDTASK

```
TimedTask;
```

A CYCLIC task and a SOFTWIREINTERRUPT task can change to a TIMEDINTERRUPT task by means of the standard procedure TIMEDTASK. Before changing the task type to TimedInterrupt, the interval time must be selected, TimedInterruptTime(time), or a default value of 255 seconds is used. Changing the task type, will not generate a ChangeTask and the task continues in the next statement.

Example:

```
TimedTask;
```

34.1.54 UPDATE

```
Update([peninfo,] variable: size [: format] [: UpdateValid]);
```

This is a very powerful procedure, used to change a variable from the keyboard. It combines the possibility to show the current value for a variable on the screen and to assign a new value to this variable from the keyboard. The variable can be declared as an internal variable in the controller, or it can be declared to be located on the P-NET network.
It is only possible to change, update, a variable if the cursor is inside the field on the screen where the variable is shown. If the cursor is not inside the field, the variable can not be changed from the keyboard and **Update** operates like the standard procedure **Display**.

Update can be used on simple types and string types. The variable is shown with the reference point for the first character in (peninfo.x, peninfo.y).

If peninfo is omitted, then the pen variable within the block called DefaultPen, is used.

If the variable is a **string type**, the parameter SIZE denotes the maximum **field width, in pixels**, for representing the string on the screen. If the field width is larger than the actual string, the remaining field is filled with blank pixels, otherwise the string is written until the maximum field width is exceeded. When the procedure is done, PenInfo.X has been moved SIZE pixels to the right.

If the variable is a type different from string, SIZE denotes the **number of characters** that is written for the variable. The parameter FORMAT is a value for representing the information on the screen. When the procedure is done, peninfo.x has been moved (size * (width of one digit)) to the right (peninfo.x is pointing at the first pixel after the field).

If variable is of type TIMER, REAL or LONGREAL, format has the following meaning:

0-..	Number of digits to the right of the decimal point. 2 is the default value.
-1	The variable is represented with floating-point.
-2	The variable is represented with exponent. For the type TIMER or REAL the exponent is always 2 digits and a sign. For the type LONGREAL the exponent is always 3 digits and a sign.

If the variable is of type BYTE, WORD INTEGER or BOOLEAN, format has the following meaning:

0	Decimal representation. This is the default value.
-3	Hexadecimal representation.
-4	Binary representation.
-5	Decimal representation with leading zeros.

If the variable is of type char or byte, format additionally has the following meaning:

-6	ASCII representation.

If the variable is of type string, format is not used.

UpdateValid is a boolean or a booelan expression. If *UpdateValid* is ON, the variable can be changed from the keyboard. If *UpdateValid* is OFF, the procedure operates like the procedure **Display**. *UpdateValid* can be any boolean or boolean expression defined by the user and it is independent of the cursor position. Default value for *UpdateValid* is ON.

Examples:

```
(* b is a BYTE, CharVal is a CHAR, r is a REAL *)
(* Str is a string[20] and PassWordOK is a boolean *)
(* This gives the following representation:  *)

b:=255;
Update(b:3:0);   (* 255 *)
Update(b:4:-3);  (* 00FF *)
Update(b:8:-4);  (* 11111111 *)

Update(CharVal:1:-6);
(* the first character from the Inputstring is moved directly to the variable
   without any conversion *)

r:=12.345678;
Update(r:7:2);   (*   12,34 *)
Update(r:7:-1);  (* 12,3456 *)
Update(r:7:-2);  (* 1,2e+01 *)

Update(Str:120:PassWordOK);
```

34.2 Standard functions

34.2.1 BITTEST

```
BitTest(Error [,errorbit, .., errorbit]):BOOLEAN
```

or

```
BitTest(Transmission, TransmissionErrorBit):BOOLEAN
```

Bittest is a function used for testing error bits, generated by the automatic error detection system (see also WHEN ERROR, CLEAR(ERROR), DISABLE(ERROR), ENABLE(ERROR) and RAISE(ERROR)) in the P-NET operating system. The function returns a boolean.

BitTest(Error [,errorbit, .., errorbit])
Using Bittest on ERROR allows you to test error bits, generated by the automatic error detection system. If the bit specification is omitted, Bittest is true if one of the errors are true, otherwise the specified errors are tested.

The different error bits to test are:

```
PnetError,    HisError,ModuleError,  ActError,DataError,
BufferError,  ArithmicError,         IndexError,          ConvertError,
```

Example:

```
IF BitTest(Error) THEN ErrorFound:=TRUE;
(* test if any error bits are set *)
```

```
IF Bittest(Error, IndexError, BufferError) THEN
    InternalError:=TRUE;
(* test if the error was caused by an index error or a buffer error *)
```

BitTest(Transmission, TransmissionErrorBit)
Using Bittest on TRANSMISSION allows you to test error bits, generated by the P-NET operating system. Bittest is true if the corresponding error bits are true. TransmissionErrorBit is a mask (16 bits) where the error bits corresponds to the bits in the fieldvariable ErrorCode from the InterFaceErrorBuffer (see the WHEN ERROR chapter).

Example:

```
IF BitTest(Transmission,$0020) THEN
  ShortCircuitError:=TRUE;
(* test if the P-NET is short-circuited *)
```

34.2.2 BUFFEREMPTY

```
BufferEmpty(buffername):BOOLEAN
```

Before a buffer is assigned to a variable, the program must check if the buffer is empty. This is done with BufferEmpty(buffername). The function returns a boolean, TRUE if the buffer is empty. If an empty buffer is assigned to a variable, an error is produced, and the value of the variable will be undefined.

Example:

```
IF BufferEmpty(KeyboardBuffer) THEN ChangeTask
```

34.2.3 BUFFERFULL

```
BufferFull(buffername):BOOLEAN
```

Before a variable is assigned to a buffer, the program must check if the buffer is full. This is done with BufferFull(buffername). The function returns a boolean, TRUE if the buffer is full. If a variable is assigned to a buffer and the buffer is already full, an error is produced, and the value will not be stored into the buffer.

Example:

```
While BufferFull(KeyboardBuffer) DO ChangeTask;
```

34.2.4 CONVERT

```
Convert(variable):integer_type
```

or

```
Convert(variable):boolean_array_type
```

A special typecasting can be performed for integer types to boolean array types and visa versa through a CONVERT function. The CONVERT function is called with a variable of the type that must be converted and then the function transforms it to the result type.

If the function is called with an integer type, then the result type must be a boolean array type with a size in byte corresponding to the integer type.

If the function is called with a boolean array type, then the result type must be an integer type with a size in byte corresponding to the boolean array type.
NOTE: the boolean array must start with index 0.

Example:

```
 VAR
     Bit8Var  : ARRAY[0..7] OF BOOLEAN;
     Bit16Var : ARRAY[0..15] OF BOOLEAN;
     ByteVar  : BYTE;
     IntVar   : INTEGER;

BEGIN
ByteVar:=Convert(Bit8Var);
     (* convert a 8 bit boolean array to a byte *)

Bit16Var:=Convert(IntVar);
     (* convert an integer to a 16 bit boolean array *)
```

34.2.5 CURSORINWINDOW

```
CursorInWindow(peninfo):BOOLEAN
```

CursorInWindow checks if the cursor is found within a certain window. The window number is specified in the pen denoted by peninfo. If the cursor is found within the window, the function returns TRUE, otherwise FALSE is returned. The procedure can only be used for PD5020.

Example:

```
IF CursorInWindow(AlarmPen) THEN
    SetCursorType(AlarmPen,NormalCursor);
      (* check if the cursor is found within the window denoted by the
      windowno. that is found in AlarmPen *)
```

34.2.6 CURSORWITHIN

```
CursorWithIn([peninfo,] Width, Height):BOOLEAN
```

CursorWithIn checks if the cursor is found within a certain field. The upper left corner of the field is specified by peninfo. The size of the field (width and height) is defined in pixels. If the cursor is found within the field, the function returns TRUE, otherwise FALSE is returned.

Example:

```
IF CursorWithIn(MyPen,20,20) THEN Found:=TRUE;
    (* check if the cursor is found within a field of size 20 by 20 pixels *)
```

34.2.7 MYTASKNO

```
MyTaskNo:INTEGER
```

This function returns the task number for the task calling the function. The function is an integer type.

Example:

```
Display(MyTaskNo:3:0);
```

34.2.8 POINTEROK

```
PointerOK(ptr):BOOLEAN
```

This function is used to test if a pointer is set to point at a variable of the right type. The function returns TRUE if Ptr is valid. The function is a boolean type.

Example:

```
IF NOT PointerOK(MyPtr) THEN MyPtr -> MyDefaultValue;
```

34.2.9 POINTERTONODE

```
PointerToNode(Node, SWNo [, Offset [, BitNo]])
```

When you want to access variables which are not declared within the controller, PointerToNode is used to set a pointer to point at the variable specified by the parameters in the function call. Node denotes an index for a node element in the NodeList, which specifies the node address for the module and the module type. SWNo denotes the Softwire number for the variable you want to access in the module specified by Node. Offset denotes an offset in bytes if you access a complex variable. BitNo denotes a bit number, calculated from the Offset. See the chapter ACCESSING NOT DECLARED VARIABLES for further information.

Example:

```
MyPtr-> PointerToNode(NodeNo, SoftwireNo, IndexNo*4);
```

34.2.10 STRVAL

```
StrVal(str[:mode])
```

This function converts the value of string type expression, STR, to its numeric representation. MODE denotes the format in which the string is represented and it is an integer type. The values for Mode is described below. If Mode is omitted, the default value is set to 0. If the character sequence in the string is illegal according to the specified mode, an error is generated (ConvertError) and the result is stored as 0 (zero). If the string represents a real value including decimal point, the character for decimal point must correspond to the selected CountryCode (please refer to the manual for the controller in question for further details on CountryCode). Otherwise an error is generated. The function type is the same type as the left side of the statement or the same type as the other operands in the expression.

If the result type for the function is of type TIMER, REAL or LONGREAL, mode is not used.
If the result type for the function is a simple type different from TIMER, REAL or LONGREAL, mode has the following meaning:

0	The string is in decimal representation with leading spaces.
-3	The string is in hexadecimal representation.
-4	The string is in binary representation.
-5	The string is in decimal representation with leading zeros.

Example:

```
RealRead:=StrVal(LoadString);
(* convert a loaded string from external equipment to a real value *)
```

34.2.11 TAB

```
Tab(Position [,Char]):STRING
```

This function is used to fill a specified character into a string to a selected position in the string. If the string length is less than POSITION, the string will be appended the character CHAR until the string length is equal to POSITION, otherwise the function does nothing. If CHAR is omitted, a space character is taken as default.

Example:

```
Str:='Setpoint ' + Tab(25,'.') + SetPoint:6:1 + ' kg';
```

If SetPoint is equal to 135.2 kg, Str will be as follows:

```
Str = 'Setpoint ..... 135.2 kg'
```

34.2.12 TESTANDSET

```
TestAndSet(bool):BOOLEAN
```

This function is used to test a boolean value, bool, and, if the boolean value is FALSE, set the boolean to TRUE. If the boolean value is TRUE already, the function returns TRUE and the boolean is not affected. The function returns the value of the boolean as a result of the TEST part of the function. This function can be used to test a variable to see if it is FALSE (free), and if so, then set it to TRUE (reserve it), all in one instruction. This facility is very useful in multitasking systems when more tasks have access to the same variables. BOOL must be a global variable. The function is a boolean type.

Example:

```
While TestAndSet(PrinterReserved) DO ChangeTask;
    (* Wait until the printer is available and then reserve it *)
```

34.2.13 VAL

```
Val(x)
```

This function is used to change the value of the expression x to another type. The expression x must be an ordinal type. The function type is the same type as the left side of the statement or the same type as the other operands in the expression.

Example:

```
TYPE
   ColourType = (Red, White, Green, Blue, Black, Yellow);
VAR
   Colour : ColourType;
BEGIN
   Colour:=Val(4);          (* Colour is set to Black *)
```

34.2.14 VARNAME

```
VarName(SoftWireNumber, [Offset]):STRING
```

This standard function is used in connection with the automatic error detecting system and it returns the stringconstant which is declared after NAME in the global variable declaration part. The function is called with a number, the softwire number for the variable. The fields ModuleSWNo and VarSWNo in an element from the InterFaceErrorBuffer holds the softwire numbers for the variables which caused the error. If the variable at the softwire number has been declared without a NAME, the function returns an empty string.

OFFSET is used to get NAME defined for a channel in an interface module, with the channel number as offset. Default value is 0.

Example:

```
ErrorString := 'Error in ' + VarName(ErrorBlock.SWNo);

ChannelNo:= ErrorBlock.VarAddr DIV $10;
ErrorText:= VarName(ErrorBlock.SWNo, ChannelNo);

Str := VarName(ErrorBlock.SWNo);
```

34.3 Standard constants

OFF has the same meaning as **FALSE**.

ON has the same meaning as **TRUE**.

MAXINT = 32767, the maximum integer value.

NIL is a constant for a pointer. A pointer-value set to NIL does not point to anything.

35 Comparing Process-Pascal ver. 4.0 to ISO 7185 Standard Pascal.

This list compares Process-Pascal to ISO 7185 STANDARD PASCAL as defined in the book **PASCAL USER MANUAL AND REPORT** THIRD EDITION by Kathleen Jensen and Niklaus Wirth (published by Springer-Verlag).

35.1 Exceptions to ISO 7185 STANDARD PASCAL.

In ISO 7185 STANDARD PASCAL, an identifier can be of any length and all characters are significant. In Process-Pascal, an identifier can be of any length, but only the first 100 characters are significant.

In ISO 7185 STANDARD PASCAL, a comment can begin with { and end with *), or begin with (* and end with }. In Process-Pascal, comments must begin and end with the same set of symbols.

In ISO 7185 STANDARD PASCAL, it is an error if the value of the selector in a CASE statement is not equal to any of the case constants. In Process-Pascal, this is not an error; instead the CASE statement is ignored unless it contains an ELSE clause.

In ISO 7185 STANDARD PASCAL, statements that threaten the control variable of a FOR statement are not allowed. In Process-Pascal, this requirement is not enforced.

ISO 7185 STANDARD PASCAL can operate on files. It is not possible to operate on files in Process-Pascal and in that reason the following procedures are not implemented:

Pack	Unpack
Read	Readln
Write	Writeln
Eof(f)	Eoln(f)
Get(f)	Put(f)
Reset(f)	Rewrite(f)
Page(f)	

ISO 7185 STANDARD PASCAL can operate with pointers. It is not possible to use dynamic pointers in Process-Pascal and in that reason the following procedures are not implemented:

Dispose(q) New(p)

ISO 7185 STANDARD PASCAL can operate with recursive procedures and functions. It is not possible to use recursivity in Process-Pascal.

In ISO 7185 STANDARD PASCAL, some arithmetic functions are available. The following functions are not available in Process-Pascal:

Arctan(x)	Exp(x)
Ln(x)	Sin(x)
Sqr(x)	Sqrt(x)

These functions can be written in Process-Pascal by using series. A number of these functions can be found in the Application folder in a file called MATH.INC.

In ISO 7185 STANDARD PASCAL, the WITH statement can be used. This statement is not implemented in Process-Pascal.

Conformant array schemes are not supported by Process-Pascal.

35.2 Extensions to ISO 7185 Standard Pascal.

Process-Pascal is integrated with P-NET, a local area network, which allows use of distributed data. Process-Pascal is specially designed for multitasking.

Process-Pascal implements the additional integer types LONGINTEGER, BYTE and WORD and the additional real type LONGREAL.

Process-Pascal implements the additional type TIMER, which is assign compatible with the type REAL. A variable of type TIMER will count down in real time when assigned a value.

Process-Pascal implements the additional type BUFFER, which, like an ARRAY type, has a fixed number of components of one type. A BUFFER is accessed only by the buffers identifier without any indexes.

Process-Pascal implements the additional types VIDEOBITMAP, LARGEBITMAP and SMALLBITMAP.

Process-Pascal implements string types, which differ from the packed string types defined by ISO 7185 STANDARD PASCAL in that they include a dynamic-length attribute that can vary during execution.

String constants are compatible with the Process-Pascal string types, and can contain control characters and other nonprintable characters.
String-type variables can be indexed as arrays to access individual characters in a string.

The relational operators can be used to compare strings.

Process-Pascal implements typed constants, which can be used to declare initialised variables of all types.

Variables can be declared at absolute memory addresses using an AT ADDRESS clause.

Constant, type, variable, procedure and function declarations can occur any number of times in any order in a block.

An identifier can contain underscore characters (_) after the first character. Integer constants can be written in hexadecimal notation; such constants are prefixed by a $.

The type of an expression can be changed to another type through a value typecast.

The CASE statement allows constant ranges in CASE label lists, and provides an optional ELSE part.

35.3 Standard Procedures and Functions

Process-Pascal implements the following standard procedures and functions, which are not found in ISO 7185 STANDARD PASCAL:

AlarmHornOnOff
AlarmPulseOn
And
BitTest
Box
BoxTo
BufferEmpty
BufferFull
ChangeTask
Clear
ClearWindow
CloseWindow
ContinueTask
ContrastControl
Convert
ConvertErrorCode
CursorInWindow
CursorTo
CursorToAbs
CursorWithin
CyclicTask
Disable
Display
DisplayOnOff
Enable
InitBuffer
InitPort
InitPort1
InterruptTask
LedOnOff
LightControl
LightOnOff
Line
LineTo
MaxRunTime
MoveCursor
MovePen
MySWNo
MyTaskNo
OpenWindow
Or
PCodeCall
PenRefTo
PenTo
PenToAbs
PerformUpdate
PointerOk
PointerToNode
Raise
RestartTask
RetryIfLegal
Return
SetCharacterGenerator
SetColors
SetCursor
SetCursorColors
SetCursorType
SetInputString
SetScreen
SetVideo
SetWindow
SetWindowFrame
StopTask
StrVal
SystemCall
Tab
TestAndSet
TimedInterruptTime
TimedTask
Update
Val
Varname
ZoomIn
ZoomInHor
ZoomOut
ZoomOutHor
ZoomOutVer

35.4 Reserved words in Process-Pascal

ADDRESS
AFTER
AND
ARRAY
AT
BEGIN
BITMAP
BOOLEAN
BUFFER
BYTE
CASE
CHANNEL
CHAR
CONFIG
CONST
CYCLIC
DEFINE
DIV
DO
DOWNTO
ELSE
END
ERROR
EXTERNAL
FALSE
FOR
FORWARD
FROM
FUNCTION
GOTO
IF
IMPORT
IN
INITIALIZE
INTEGER
INTERCOM
INTERFACE
INTERNAL
INTERRUPT
LABEL
LARGEBITMAP
LONGINTEGER
LONGREAL
LOOP
MAXINT
MOD
MODULE
NAME
NET
NIL
NOT
OF
OFF
ON
OR
PLACE
POINTER
PROCEDURE
PROGRAM
READY
REAL
RECORD
REPEAT
RETURN
RUNTIME
SET
SMALLBITMAP
SOFTWIRE
SOFTWIREINTERRUPT
STRING
SUSPENDED
TASK
THEN
TIMEDINTERRUPT
TIMER
TO
TRUE
TYPE
UNTIL
UNUSED
VAR
VIDEOBITMAP
WHEN
WHILE
WITH
WORD

35.5 Compiler directives

A compiler directive controls some of the compilers features and are introduced as comments with a special syntax. Process-Pascal allows compiler directives wherever comments are allowed.

A compiler directive starts with a $ as the first character after the opening delimiter. The $ is immediately followed by a letter that designates the particular directive.

{$L-}
LISTING OFF
This directive is a switch directive that turns OFF the listing of the source file and error messages in the .LST file.

{$L+}
LISTING ON
This directive is a switch directive that turns ON the listing of the source file and error messages in the .LST file.

{$I'filename'}
INCLUDE FILE
This directive instructs the compiler to include the named file in the compilation. The file is inserted in the compiled text after the directive. If **filename** does not specify a directory, then the current directory is searched first, and if not found, the default directory is searched.

There are no restrictions to the use of include files. This means that an include file can be specified in the middle of a statement part.

Process-Pascal allows, at most, five input files to be open at any given time. This means that include files can be nested up to five levels deep.

{$P=nn}
LINES per PAGE in LST FILE
This directive determines how many lines per page there shall be in the .LST file. The default value is 60 lines per page. **nn** is an integer value.

{$MIB property}
MIB PROPERTY
This directive is used to set one or more default properties to be used in VIGO. The Compiler can automatically generate a SMB file that holds a description of all constants and variables declared in the Process-Pascal program. Each constant and variable has it's own set of properties to describe visibility, backup requirement, simulation, permission for read access, permission for write access or protected write access.

The following properties are available:

MIB_Invisible	MIB_Visible
MIB_NoBackup	MIB_Backup
MIB_Simulation	MIB_NoSimulation
MIB_NoReadAccess	MIB_ReadAccess
MIB_NoWriteAccess	MIB_WriteAccess
MIB_NoProtectedWriteAccess	MIB_ProtectedWriteAccess

The *property* following the $MIB directive can be one or more of the above properties in any combination.

Example:

```
{$MIB MIB_Visible, MIB_Backup }
```

This will set the default values for MIB properties for the succeeding declarations to be Visible and with the Backup property set.

36 Restrictions in Using Process-Pascal.

When programming a controller, the following restrictions must be considered:

When using a controller with a display unit, a cursor must always be defined in the program to avoid unexpected flicker on the screen.

If a variable of the type BUFFER or TIMER is a component of a complex variable, this component variable is only to be used internally in the controller. (P-NET restriction).

When one variable is assigned to another variable, at least one of the variables must be declared internally in the controller. (P-NET restriction).

It is not allowed to use recursive procedures/functions.

Using UPDATE on a variable declared in a procedure (local in a procedure) is not allowed, but it can be detected, either by the compiler or by the operating system. If it is detected by the operating system, an error is generated (Update not allowed).

If a local variable, declared in a **global** procedure, is passed to another **global** procedure as a VAR parameter, and the variable is UPDATE'ed, then either the compiler, nor the operating system can detect the failure and the result of the performed update is unpredictable.

If you access an external variable (via P-NET and a softwire number), this softwire number must not be declared as an indirect array variable.

If a Net address is declared to be a string, the string must not be declared as an indirect variable pointing to another string.

Interrupt on indirect variables is not allowed.

There is a 32 Kb limit for one variable.

A STRING can't be appended to another string and stored to itself, example:

```
Str:='str2';
Str:='text' + Str;
```

then Str = 'texttext', but it really should be 'textstr2'.

If you compare two BOOLEANs and one is a part of a variant record, which shares the memory location with e.g. a BYTE, a COMPARE will only result in TRUE if the BYTE values are equal.

Please note that the error codes in PD 5000 are not the same as in PD 4000 and the use of the Bittest function is therefore not compatible between the two controllers.

When you use a TIMEDINTERRUPT task with an interrupt time set to less than 2 sec. to access data on another network through a gateway controller, this may cause the controller to behave “slow” in case of transmission errors in the “reply-request” issued by the gateway controller. Instead you should use a cyclic task.

It is not possible to declare Constant strings that include a null character. If you need a string like e.g. UnderOff = #$1B#$2D#$00 (esc-0), you must declare UnderOff as a variable and then assign each character to the string by means of the Char function.

37 Error Messages and Codes

0: Illegal character
An illegal character was found in the source text. You may have forgotten the quotes around a string or a character different from a letter or a digit or an underscore has been used.

1: Digit expected
A digit does not appear where it should. A location in a variable declaration has been mistyped or a real value holds a non digit character.

2: Hex digit expected
A hexadecimal digit does not appear where it should. A location in a variable declaration followed by a $ sign has been mistyped.

3: Exponent too big
The exponent is too big in a constant declaration or in a constant assignment.

4: Hex overflow
A constant assignment for an integer type preceded by a $ sign is too large.

5: Long overflow
A constant assignment for a longinteger type is too large.

6: String too long
The maximum length of a string must not exceed 255 characters.

7: String crossing lineend
The declaration of a string must be hold within one line. You might have forgotten the ending quote in string constant.

8: File ended unexpected
The source file ended before the final END for the program or module. You might have a comment which is not closed.

9: Delimiter expected
A delimiter does not appear where it should. An operator may be missing or a comma may be missing in a multidimensional array assignment.

10: Semicolon expected
A semicolon does not appear where it should.

11: Colon expected
A colon does not appear where it should.

12: '[' expected
A left bracket does not appear where it should.

13: ']' expected
A right bracket does not appear where it should.

14: '(' expected
An opening parenthesis does not appear where it should.

15: ')' expected

A closing parenthesis does not appear where it should.

16: Identifier expected
An identifier was expected here. You may have forgotten the identifier in a declaration or in an assignment.

17: 'OF' expected
The reserved word OF does not appear where it should.

18: Subrange expected
A subrange does not appear where it should. You might have forgotten to denote the subrange in an array declaration.

19: '=' expected
An equal sign does not appear where it should.

20: Integer expected
An integer does not appear where it should. May be you have declared a softwire interrupt task connection with an identifier or a letter.

21: 'PROGRAM' expected
The reserved word PROGRAM does not appear where it should.

22: '.' expected
A period does not appear where it should.

23: 'BEGIN' expected
The reserved word BEGIN was expected here. You may have forgotten the word BEGIN or there is an error in the block structure, i.e. an END is missing.

24: 'END' expected
The reserved word END was expected here. You may have forgotten the word END or there is an error in the block structure, i.e. a BEGIN is missing.

25: error in type
This symbol can not start a type definition.

26: error in constant
A constant does not appear where it should.

27: Identifier declared twice
The identifier has already been declared once within the current block.

28: ':' or '->' expected
In the global variable declaration a ':' or a '->', an indirect variable, was expected after the identifier.

29: Address expected
A global variable was declared at a fixed address but no absolute address appears where it should.

30: ObjectType for program expected
The program header must specify an object type. The object type must be in square brackets.

31: Identifier not a constant
A subrange or size specifier (in array or buffer declarations) is not a constant.

32: Identifier not a type
The identifier does not denote a type as it should.

33: Field identifier expected
A record variable is used and the identifier does not denote a field within that record. You might have forgotten the field identifier or spelled it wrongly.

34: Comma expected
A comma does not appear where it should.

35: Error in label
May be you have declared a label with a negative value or a GOTO label statement denotes a label which is not declared. A label declaration must be a positive integer value.

36: Record field expected
A record structure is accessed but the field identifier is not found in the record definition. May be you have misspelled the field identifier.

37: Different types
A variable and the expression in an assignment statement are of incompatible types, or the actual and the formal parameters in a call to a procedure or a function are of incompatible types, or the operands in an expression are of incompatible types.

38: Variable not declared
An indirect variable declaration points to a variable which is not declared.

39: Illegal constant
The index specifier in an array declaration is illegal. The index specifier must be a constant identifier or an ordinal value. You might have used a reserved word as index specifier.

40: Range error
A constant index for an array is outside the specified range for the array.

41: Too many index
An array is accessed with too many index's.

42: Identifier not declared
The identifier is not declared within the scope for the current block.

43: Illegal Range
The specified range in an array declaration is not legal. Only ordinal values must be used.

44: Illegal type
The type of the operand in an expression is not legal or a variant part is stated in an interface declaration.

45: "-> PointerToNode" only allowed with a POINTER
The PointerToNode function can only be used with a pointer type, .e. the left side of the assignment must be a pointer type.

T

46: Illegal directive
The compiler directive letter is unknown.

47: Variable or ARRAY expected
An indirect variable declaration is not succeeded by a variable identifier or the reserved word ARRAY.

48: Illegal identifier
The used identifier is not legal here. You may have used a wrong identifier in an interface declaration or in connection with disable/enable ERROR.

49: Block symbol expected
A block symbol is expected here. A statement is maybe found outside a block. You might have an END to much. A block symbol can be: program, module, const, type, var, label, begin, procedure, function, task.

50: Indirect VAR not allowed to a pointer type
It is not allowed to create an indirect variable that points to a pointer type.

51: Standard name not allowed
You are trying to re-declare a standard identifier.

52: Identifier is not an array
The identifier is not an array. You might try to use index access to a simple variable.

53: ':=' expected
An assignment operator does not appear where it should.

54: TO or DOWNTO expected
The reserved word TO or DOWNTO does not appear where it should.

55: DO expected
The reserved word DO does not appear where it should.

56: Statement expected
A statement is expected here but a block symbol or a reserved word may be found instead.

57: Label expected
The identifier does not denote a label as it should.

58: THEN expected
The reserved word THEN does not appear where it should.

59: UNTIL expected
The reserved word UNTIL does not appear where it should.

60: Factor expected
A factor is expected in an expression.

61: Boolean expression expected
The preceding expression must be of boolean type.

62: Type conflict
There is a type conflict in the statement. Maybe you are trying to assign a string to a real or assign a record variable to another record variable of another type.

63: Variable or function expected
Variable or function expected in an assignment statement.

64: Procedure expected
An identifier is found as a statement. A procedure was expected.

65: Types incompatible
You might have incompatible types of the variable and the expression in an assignment statement.

66: Constant not allowed
A constant is not allowed here. Maybe you are calling a procedure or function with a constant as a parameter and the procedure/function declaration specifies a VAR parameter.

67: Error in parameterlist
There is an error in the parameter list in a procedure/function call. Maybe you have too many parameters or there is an error in the types.

68: Different size
An assignment statement between to array's of different size, or a bitmap has been declared with a certain size and you are trying to insert more bytes than specified.

69: '->' Expected
An indirect array declaration is made and '->' is expected to point out each of the elements in the array.

70: Variable expected
A variable is expected here. It might be a pointer statement or a standard procedure with a parameter like TestAndSet or InitBuffer, or it could be a FOR loop where the identifier for the loop counter is not a variable.

71: Illegal interrupt number
An illegal interrupt number is used. The number must be within the range of 0..31.

72: unexpected symbol
An unexpected symbol is found where it should not appear.

73: String expected
A string is expected. You might have variable declaration with a NAME clause and the string denoting the NAME must be a constant string or a string type identifier.

74: Interface not assignable
You cannot assign an entire interface. Only one softwire number can be assigned at a time in an interface.

76: 'AT' only at global level
The reserved word AT must only be used at global level.

78: Interupt already in use
A task declaration denotes a softwire interrupt connection with an interrupt number which has already been used.

79: Illegal case type
The case variable in a case statement is illegal. Only ordinal type are allowed as case variables.

80: Case label declared twice
The case label has already been used.

81: Assignment not allowed
Assignment not allowed for constants.

83: Index error (Too high)
The size of an array or record declaration is too large.

84: Too many INCLUDE files
You have too many nested include files. The Process-Pascal compiler allows no more than 5 nested include files.

85: SimpleTypeExpected
A simple type is expected. You might have called a standard function with a complex type or declared a function to return a complex type.

86: Bitmap Expected
The standard procedure SetCursor expects a bitmap as a parameter for selecting the cursor symbol.

87: Buffer Expected
The standard procedure InitBuffer and standard functions BufferEmpty and BufferFull expects a buffer as a parameter.

88: Enumerationtype Expected
An enumeration type is expected. The result of the expression may not be an enumeration type. You might use the Val function to convert the type.

89: 'Error' Expected
The reserved word ERROR does not appear where it should. The reserved word WHEN should always be followed by ERROR.

90: Error in opening file
The specified file was not found in the current directory, nor in the directories specified by the SET PROPAS command.

91: Taskname expected
A taskname was expected here. A task declaration must hold the reserved word TASK followed by an identifier for the task name.

92: Strings must have equal length
Strings must have equal length refers to that the actual string length must not be longer than the formal parameter in a procedure or function call. The length of the actual string is allowed to be less than the length of the formal parameter.

95: Buffer not allowed
Buffer is not allowed here. You may try to assign a buffer of string type with an expression holding a number of strings.

96: Recursion not allowed
Recursion not allowed in Process-Pascal.

97: Level out of range
You have nested more than 127 procedures/functions. A procedure or function is said to be nested when it is declared within another procedure or function.

98: Version error: incompatible version of SYS-file and compiler
The Process-Pascal compiler and the SYSTEM file for the program must have the same version number.

99: Not implemented yet

100: Not allowed on this level
An Update procedure is called with a parameter which is locally declared in a procedure or function. Only global variables and variables declared in a task may be used as parameters in Update.

101: Index error
An index error has occurred. You might have declared a complex constant and are trying to assign the same index more than once.

102: Index missing
An indirect array declaration is missing an index declaration. All index's in an indirect array must point to a previous declared variable or a part of a previous declared variable with the right type.

105: Interrupt not declared
A variable has been declared with an interrupt connection but there is no task declared with the corresponding interrupt connection.

106: Softwire table overflow
The number of softwire numbers exceeds the allocated table size.

107: Local variable table overflow
The number of local variables exceeds 65.535. Maybe you should reduce the number of local variables in your program.

110: P-NET address expected
A global variable declaration denotes a net specification but the P-NET address does not appear where it should.

111: Variable has to be internal
You might have used a global variable declaration using the NAME clause, then the string identifier for the name must be declared internally in the same controller. Or you are using a string identifier for the net list, the P-NET node address, this identifier must also be declared internally.

112: Both variables may not be external
An assignment statement has external variables on both sides of the equal sign. At least one of the variables must be internal.

113: Only global variables allowed
Only global variables allowed as parameter for the standard function TestAndSet.

114: Compare two stringsconst not allowed
You are not allowed to compare to constant strings.

115: DeviceType required
A global variable declaration using the AT NET clause must include DEVICETYPE in the type declaration.

117: Return only in WHEN ERROR
The standard procedure must only be used in the WHEN ERROR part of a program.

118: Set too large
The set type is declared with more than 256 members.

119: Only allowed in interrupt task
The statement is only allowed in a softwire interrupt task. You may be trying to use the standard procedure InterruptTask in a task that was not originally declared as a softwire interrupt task.

120: Don't start your self
The standard procedure ContinueTask was called with a task identifier within the task itself.

121: 'TO' expected
The reserved word TO does not appear where it should.

122: ROM Need for this TASK too large
The code size for the task is too large. The maximum code size for a task is 64 Kbytes.

123: RAM Need for this TASK too large
The stack need for this task exceeds 32.768 bytes. The stack need consists of local variables for the task and stack for procedure and function calls. You have to split the task in more tasks or declare some of the variables as global variables.

124: INTERRUPT not allowed
Interrupt connection is not allowed on external variables, nor on local variables.

125: Illegal assignment
The assignment is illegal. Assignments can only be made to variables and pointers.

126: TASK Expected
The reserved word TASK does not appear where it should.

127: JUMP too long
Maybe you have declared a CASE structure where the code size is too large, hence the jump is too long.

128: More than one NAME for this variable
A global variable has been declared with NAME already.

129: Only allowed in string assignment
You might try to assign a non string variable to the standard function VarName. VarName returns a string.

130: Field identifier declared twice
In a record declaration you have used the same field identifier twice.

131: Memory type expected
A memory type is expected here.

132: CHAR expected
The standard function Tab expects a character as an optional parameter.

133: Timer not allowed
In a procedure or function declaration, a TIMER is not allowed as a value parameter.

134: SYS file is empty
The fixed part of the system file was not found. The fixed part begins with the type declarations for the system variables and ends with declaration of the system variable PDBoxDefinition.

135: Identifier not a routine
The identifier is not a routine.

136: Variable too large (>32KB)
The maximum size of one variable is 32 Kbytes.

137: Interrupt not connected
A softwire interrupt task is declared with an interrupt connection which is not found at the variable declaration.

138: Retry only in WHEN ERROR
The Retry statement is only allowed within a WHEN ERROR block.

139: IMPORT only at global level
You are only allowed to IMPORT compiled modules at global level.

140: Filename expected
A filename was expected here. You might try to use an include directive without

denoting a filename, or have forgotten the filename after IMPORT.

141: Memory used by another module
An IMPORT statement has already reserved the memory locations.

142: This file is not a compiled module
The file specified in an import statement is not a compiled module. You might have given in a wrong file name, maybe the source file name.

143: This file is not a symbol file
The specified file does not have the expected format.

144: Version error: incompatible version of module and compiler
The imported module has been compiled with a different version of the compiler. Re-compile the module source code with this version of the compiler.

145: Task modules only
When using the import statement after the global part it's only allowed to import task modules.

146: Identifier list not allowed
Identifier list not allowed at this place. You might try to use the AT NET or AT SOFTWIRE or an indirect declaration on a number of identifiers in a list.

147: Not allowed in Modules
You may be declaring an indirect variable within a module. This is not allowed in modules.

148: Real overflow
A constant assignment for a real type is too large. Maybe you try to set a value for timed interrupt task with a value which is too large.

149: Softwire number already used
The Softwire number specified by a PLACE clause in the global variable declaration has already been used. Increase the Softwire number, or move the declaration to an earlier part of the variable declaration.

150: File Error
A file error has occurred. The disk may be full or write access has been denied.

151: ProcVar NOT finally declared
A procedure variable has not been declared before it has been used.

152: MIB interface error. Please check version. SMB-generating disabled.
An error occurred during the SMB file generation. Check that the versions for VIGO and Process-Pascal match. No SMB file is generated.

153: MIBProperties not allowed here
The specification of MIB properties is not allowed here. MIB properties can only be specified for global constants and variables.

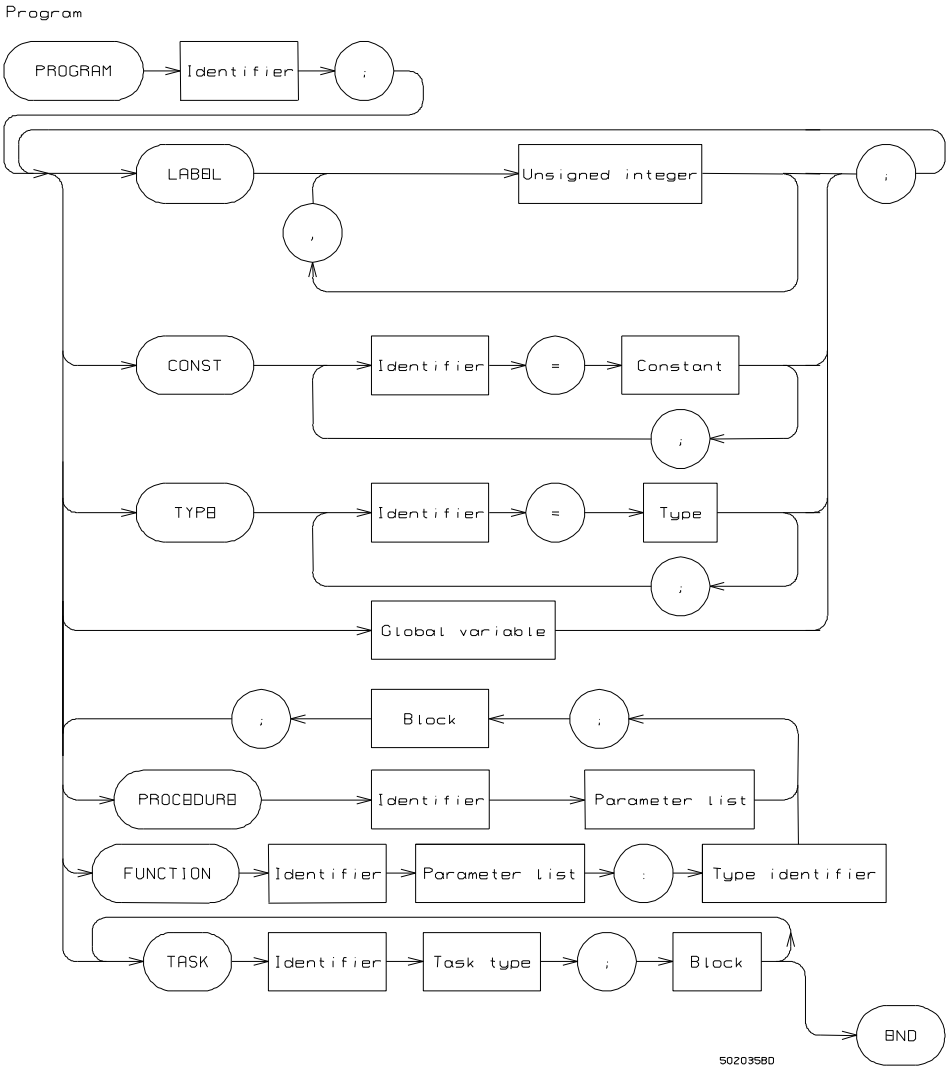
154: MIBProperties can not be changed on this variable/constant type
Some MIB properties are not allowed for certain constants and variables, e.g. you can not specify MIB_Backup for a constant.

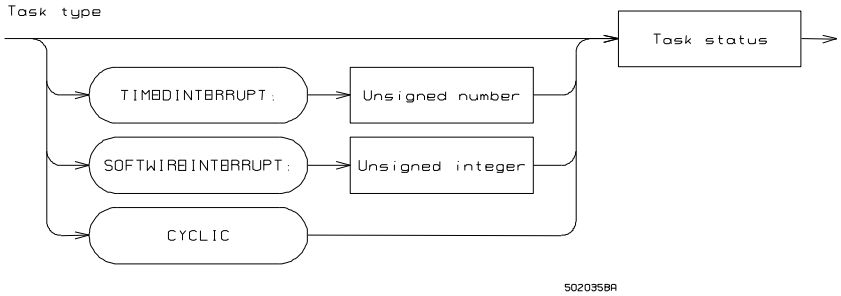
155: Compiler Error, please contact PROCES-DATA A/S
A compiler error has occurred. The compiler has come to an undefined state and compilation has stopped. Please contact PROCES-DATA A/S.

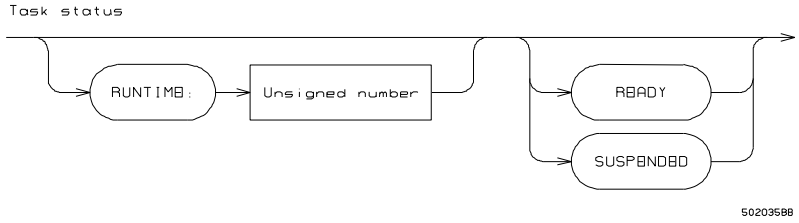
FATAL compiler error
A fatal error has occurred within the compiler. The compiler program is terminated immediately.

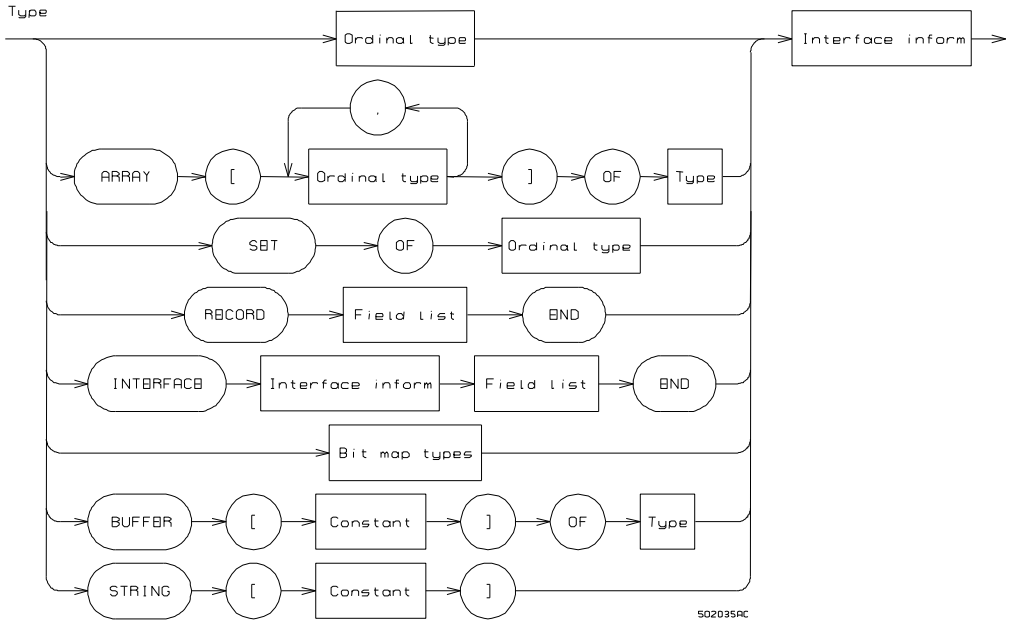
Compilation aborted at line
Compilation aborted at line xxxxx. This error is always written in connection with one of the two errors above, Symbol table overflow, or Fatal compiler error. The line number refers to a line number within the compiler. This might lead PROCES-DATA A/S to a hint on where to find the error.

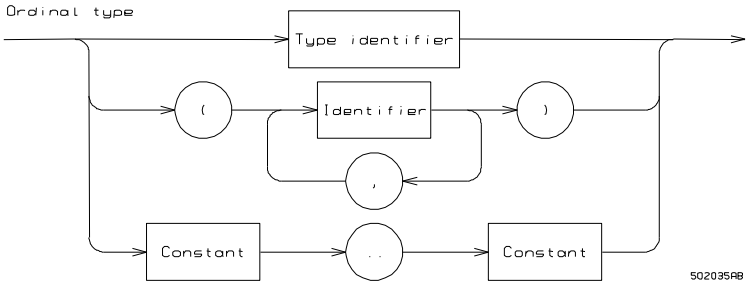
38 Syntax Diagrams

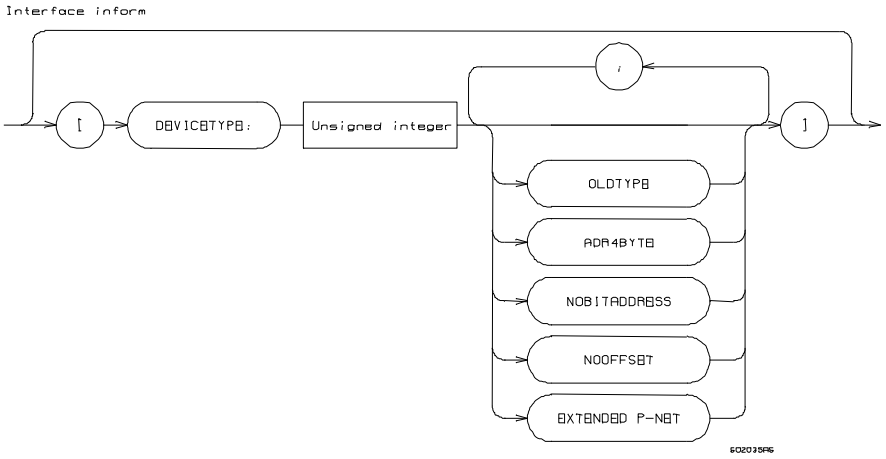


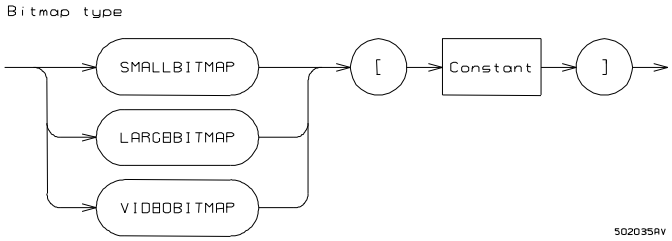


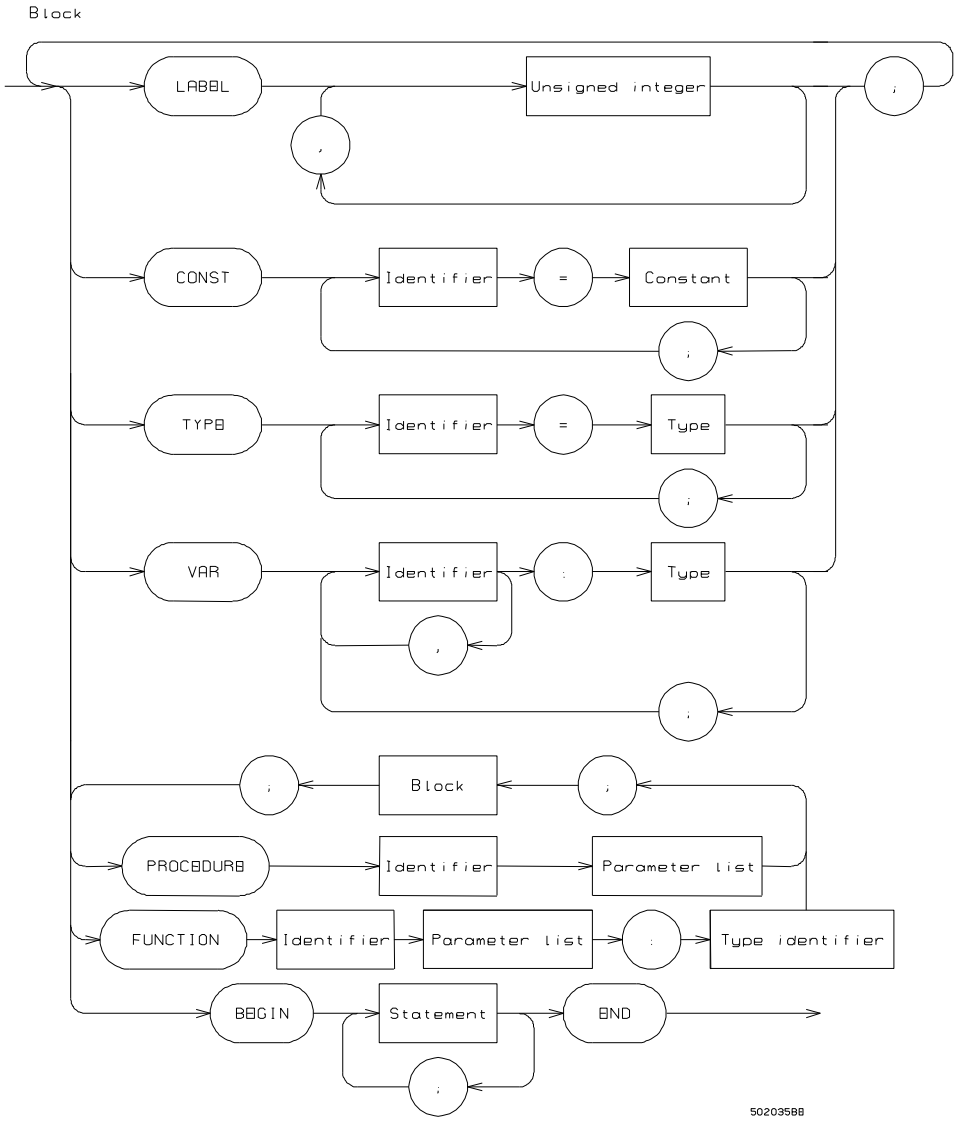

Task status
RUNTIME:
Unsigned number
READY
SUSPENDED
502035BB

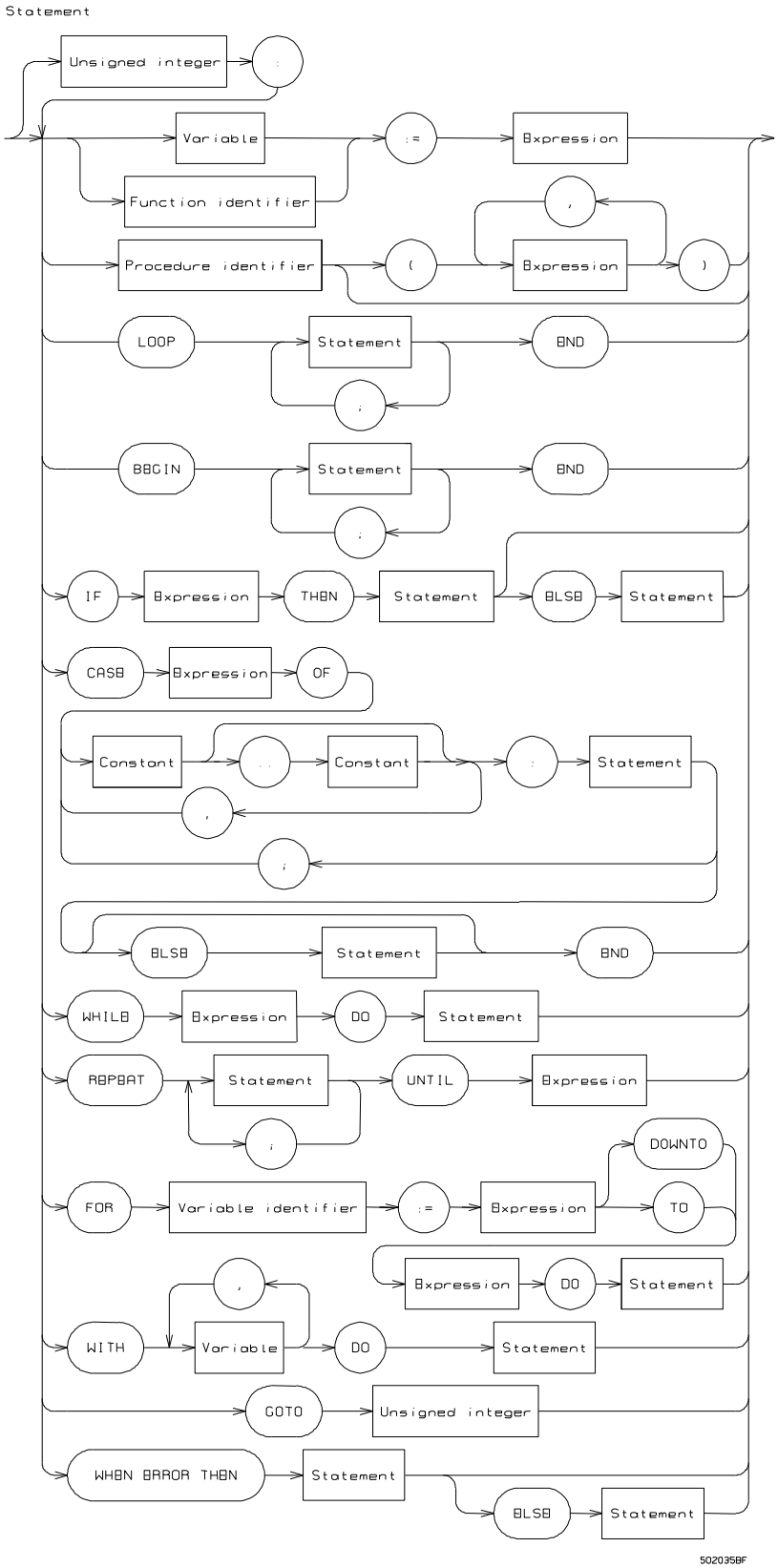

Type
Ordinal type
Interface inform
,
ARRAY
[
Ordinal type
]
OF
Type
SET
OF
Ordinal type
RECORD
Field list
END
INTERFACE
Interface inform
Field list
END
Bit map types
BUFFER
[
Constant
]
OF
Type
STRING
[
Constant
]
502035AC


Ordinal type
Type identifier
(
Identifier
)
,
Constant
..
Constant
502035AB

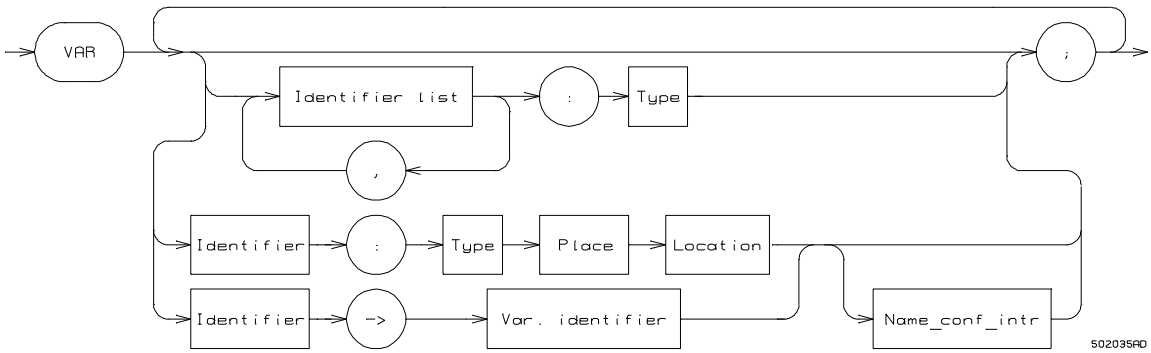

Interface inform
[
DEVICETYPE:
Unsigned integer
,
]
OLDTYPE
ADR4BYTE
NOBITADDRESS
NOOFFSET
EXTENDED P-NET
502035A6


Bitmap type
SMALLBITMAP
LARGEBITMAP
VIDEOBITMAP
[
Constant
]
502035AV

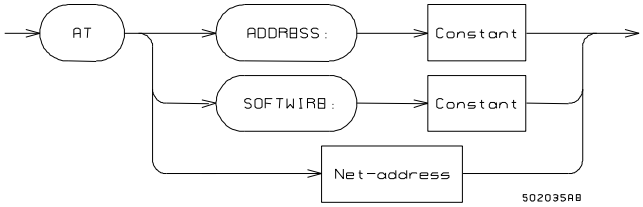

Block
LABEL
Unsigned integer
,
;
CONST
Identifier
=
Constant
;
TYPE
Identifier
=
Type
;
VAR
Identifier
:
Type
,
;
;
Block
;
PROCEDURE
Identifier
Parameter list
FUNCTION
Identifier
Parameter list
:
Type identifier
BEGIN
Statement
END
;
5020358B


Statement
Unsigned integer
:
Variable
:=
Expression
Function identifier
,
Procedure identifier
(
Expression
)
LOOP
Statement
END
;
BEGIN
Statement
END
;
IF
Expression
THEN
Statement
ELSE
Statement
CASE
Expression
OF
Constant
..
Constant
:
Statement
,
;
ELSE
Statement
END
WHILE
Expression
DO
Statement
REPEAT
Statement
UNTIL
Expression
;
DOWNTO
FOR
Variable identifier
:=
Expression
TO
Expression
DO
Statement
,
WITH
Variable
DO
Statement
GOTO
Unsigned integer
WHEN ERROR THEN
Statement
ELSE
Statement
502035BF

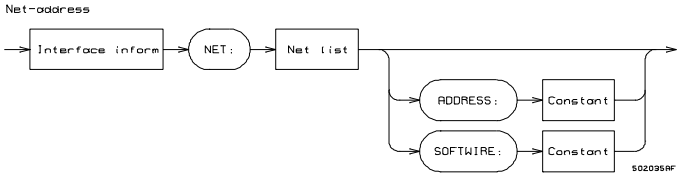
Global variable



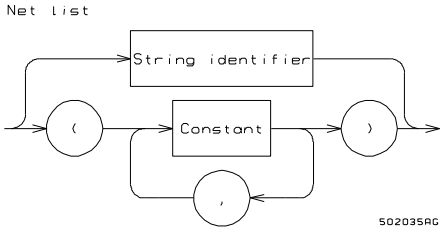
Location



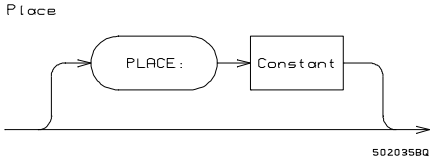
Net-address



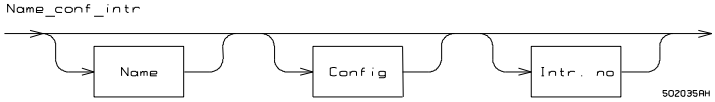
Net list

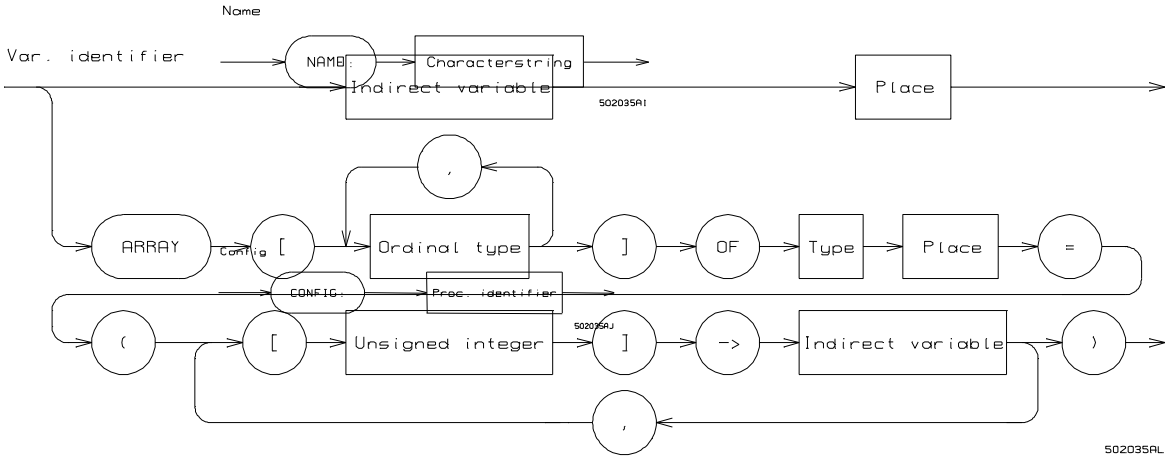


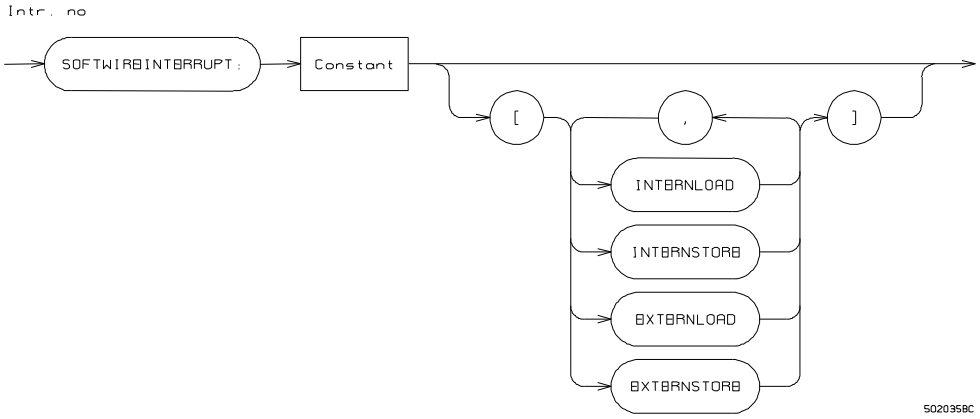
Place

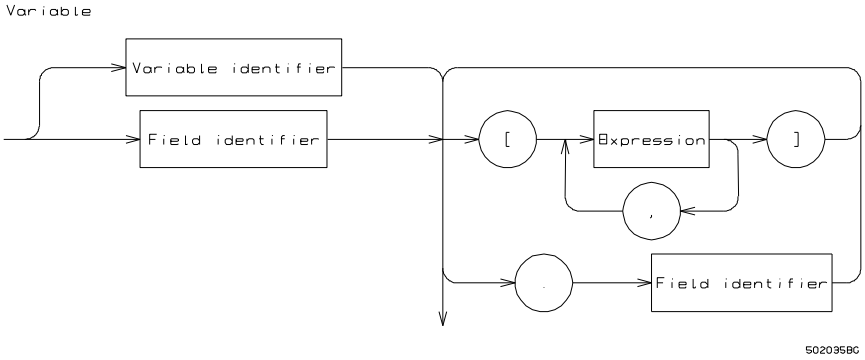


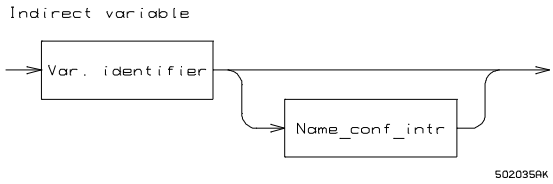
Name_conf_intr

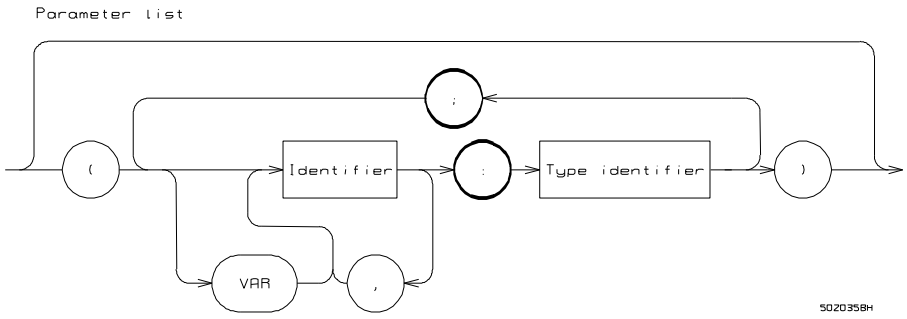


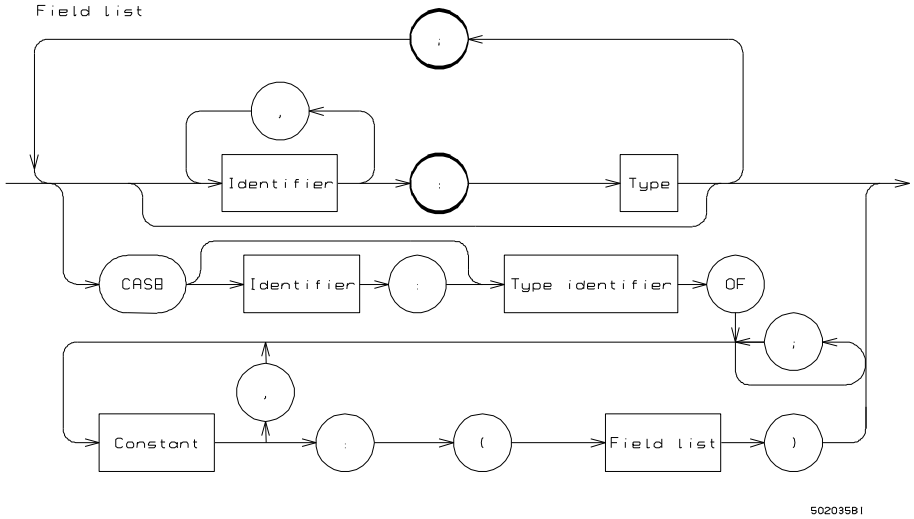

Name
Var. identifier
NAMB:
Characterstring
Indirect variable
502035A1
Place
,
ARRAY
Config [
Ordinal type
]
OF
Type
Place
=
CONFIG:
Proc. identifier
(
[
Unsigned integer
502035AJ
]
->
Indirect variable
)
,
502035AL

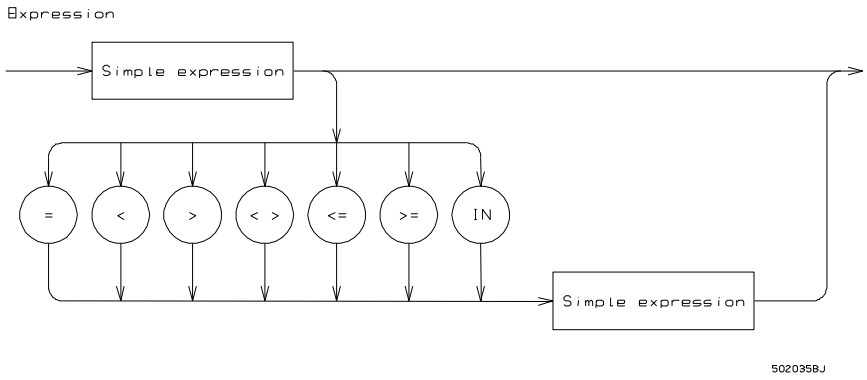

Intr. no
SOFTWIRBINTBRRUPT:
Constant
[
,
]
INTBRNLOAD
INTBRNSTORB
BXTBRNLOAD
BXTBRNSTORB
502035BC

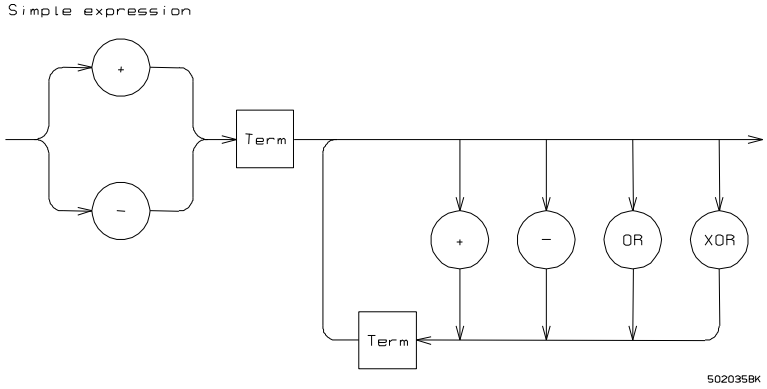

Variable
Variable identifier
Field identifier
[
Expression
]
,
.
Field identifier
502035BG


Indirect variable
Var. identifier
Name_conf_intr
502035AK

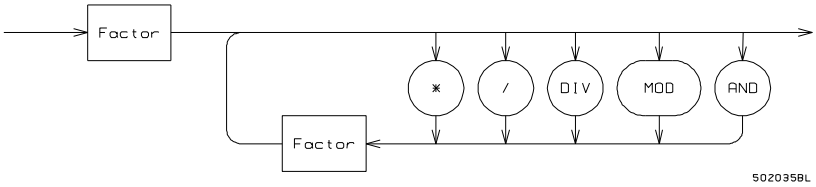

Parameter list
(
;
Identifier
:
Type identifier
)
VAR
,
502035BH


Field list
;
,
Identifier
:
Type
CASE
Identifier
:
Type identifier
OF
;
,
Constant
:
(
Field list
)
502035BI

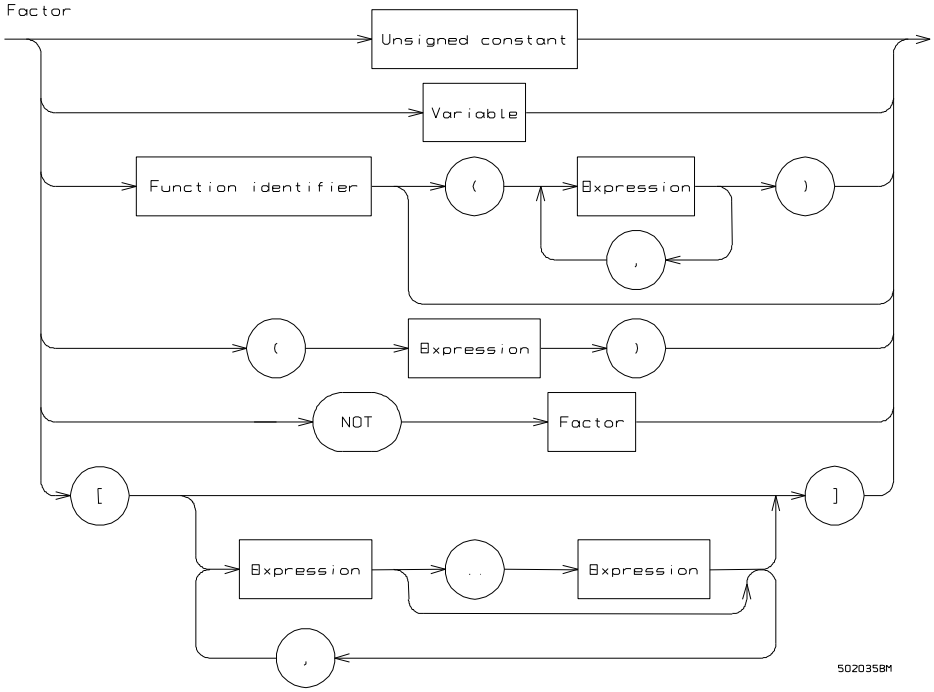

Expression
Simple expression
=
<
>
< >
<=
>=
IN
Simple expression
5020358J


Simple expression
+
-
Term
+
-
OR
XOR
Term
502035BK

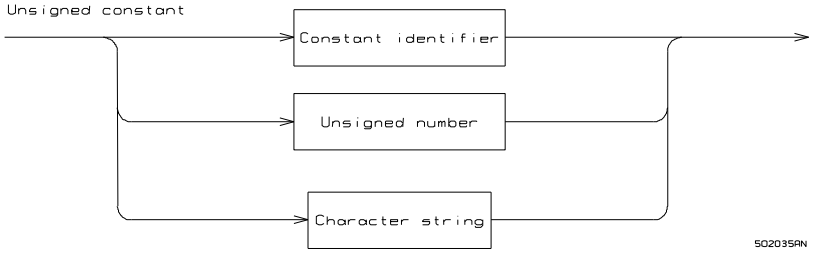
Term



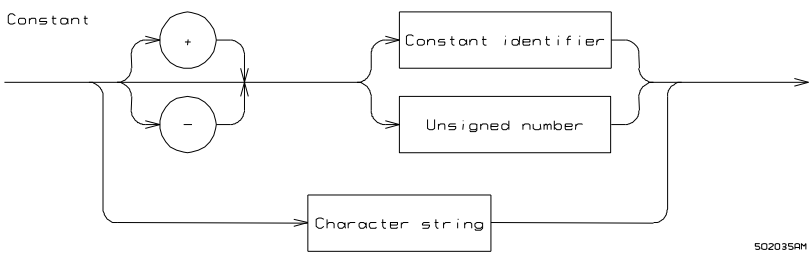
Factor

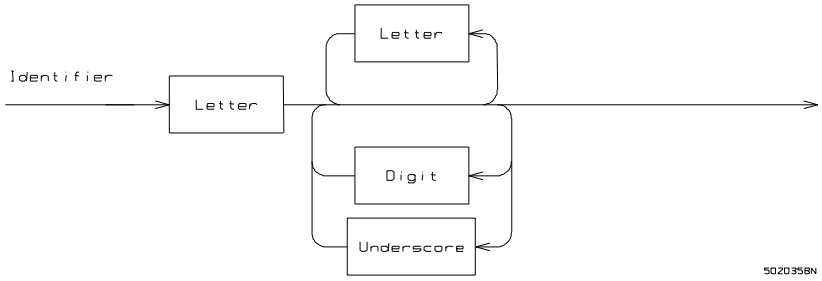


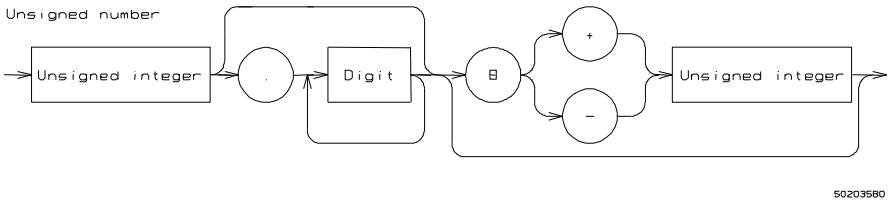
Unsigned constant

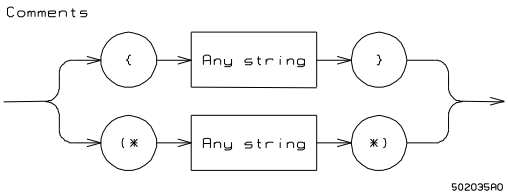


Constant




Identifier
Letter
Letter
Digit
Underscore
5020358N


Unsigned number
Unsigned integer
.
Digit
E
+
-
Unsigned integer
50203580


Comments
{
Any string
}
(*
Any string
*)
502035A0

39 Index

ActError'59;60
ActReport'61
Address'13;14
Adr2Byte'35
Adr4Byte'35
Alarm'84
AlarmHornOnOff'84
AlarmPulseOn'84
And'23
ArithmicError'59;60
Array'31
Array constant'44
Assignment'26
Begin'27
Bitmap'39
Bittest'60
BitTest'60;104
Boolean'8
Box'84;85;96
BoxTo'85
Buffer'37
    BufferEmpty'104
    BufferFull'37;105
    buffersize'37
    InitBuffer'37
BufferError'59;60
Byte'9
Calendar'73
Capabilities'75
Case'28
Changetask'4;46;49;85
Channel'35;109
Char'8
Clear'60;85
ClearWindow'86
CloseWindow'86
Color'68
    background'67
    foreground'67
    inverse'68
    transparent'68
Comments'21
Communication'77
Compiler directive'21;114
Compound statement'27
Conditional statement'27
Config'16
Constant'20
ContinueTask'53;86
Contrastcontrol'87
Convert'9;105;107
ConvertError'59;60
CountryCode'107
Cursor'65;66;70;87;94
Cursor position'69
CursorHide'70
CursorInWindow'106
CursorTo'71;87
CursorToAbs'71;87
CursorWithIn'106
Cyclic task'4
CyclicTask'52;55;87
DataError'59;60
Datamode'77
DataReport'61
DateTime'73
DefaultPen'69
Devicetype'35
Disable'54;56;60;88
Display'67;89
DisplayOnOff'90
Div'22
Downto'30
Enable'54;56;61;90
End'27
Enumerated'42
Error'60
Error
    detection'58;60;61
    ;85;86;88;91;95;9
    7;104
Error messages'117
Errorcode'63
ErrorCode'62
Expression'22;24
ExtendedPNET'36
External'81
Externload'56
Externstore'56
False'109
Field'33
For'30
Forward'54
Function'46;50
Gateway'77
Global variables'12
HisError'59;60
HisReport'61
Identifiers'6
If'27
Import'83
Include files'114
Index'31
IndexError'59;60
Indirect variable'16
Initbuffer'91
InitPort1'92
Integer'9

Interface'35;109
Interfaceerror'62
InterFaceErrorBuffer'61
Internload'56
Internstore'56
Interrupt'55;56
InterruptTask'92
Keyboard'72
KeyboardBuffer'72
Keycodes'72
Largebitmap'39
LedOnOff'92
LightControl'93
LightOnOff'93
Line'93
LineTo'93
Listing
    OFF'114
    ON'114
Local variables'12
Longinteger'9
Longreal'10
Loop'30
Maxint'109
MaxRunTime'52;94
MIB properties'115
Mod'22
Module'81
ModuleError'59;60
ModuleReport'61
MoveCursor'71;94
MovePen'69;94
MySWNo'95
MyTaskNo'106
Name'15;109
Net list'14
Nil'19;109
NoBitAddress'35
NodeAddr'74
Nodelist'75
NodeList'74
NoOffset'36
Not'23
OFF'109
Oldtype'35
ON'109
OpenWindow'95
Operator
    arithmetic'22
    logical'23
    relational'23
    string'24
Or'23
Ord'7
Parameter list'47
Pen'69
Pen position'69
PenRefTo'69;70;96
PenTo'69;96
PenToAbs'69;96
PerformUpdate'97
Pixel'39;40;69
Pixel position'69
Place'15
PnetError'59;60
PnetReport'61
Pointer'19
PointerOK'106
PointerToNode'75;107
Pred'7
Procedure'46
Procedure statement'26
Raise'61;97
Ready'53
Real'10
Real-time clock'73
Record'33
Record constant'45
Register'35
Repeat'29
Reserved words'113
Restart'54;97
Restrictions'116
RetryIfLegal'98
Return'59;98
Runtime'52
Run-time errors'58
Scope'51
ScreenInfo'65
Set'41
Setcharactergenerator'98
Setcolors'68
SetColors'98
Setcursor'66;98;99
SetCursor'70
SetCursorColors'99
SetCursorType'99
Setinputstring'99
Setscreen'66;100
Setvideo'66;100
Setwindow'66;100
SetWindowFrame'101
Simple statement'26
Smallbitmap'39
SMB file'114
Softwire'13;62;64
SoftwireInterrupt'4;52;55
SoftwireNo'62
Standard constants'109
Standard functions'104;112

Standard procedures'84;112
StopTask'53;101
String'38;89;102;107
Structured statements'27
StrVal'107
Subrange'42
Succ'7
Suspended'53
Syntax diagrams'128
Tab'107
Task declaration'52
Task status'53
TestAndSet'108
TimedInterrupt'4;52
TimedInterruptTime'54;101
TimedTask'55;102
Timer'10
To'30
Transmission'60
True'109
Type
    boolean'8
    byte'9
    char'8
    data'7
    integer'9
    longinteger'9
    longreal'10
    ordinal'7
    pointer'19
    real'10
    timer'10
    word'9
Typecasting'9;105
Until'29
Update'67;102
UpdateValid'103
Val'108
Value parameters'47
VARAddr'62
Variable declaration'12
Variable parameters'48
Variables'6
Variant part'33
Varname'108
VarName'108
VarOffset'62
Videobitmap'39;66
VIGO'75;114
When error'58;59
While'28
Window'65;67;95
    frame'95
    number'95
Word'9